NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

USE OF STATECHART ASSERTIONS FOR MODELING HUMAN-IN-THE-LOOP SECURITY ANALYSIS AND DECISION-MAKING PROCESSES

by

Michael A. Schumann

June 2012

Dissertation Supervisor: James Bret Michael

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2012	**3. REPORT TYPE AND DATES COVERED** Dissertation

4. TITLE AND SUBTITLE: Use of Statechart Assertions for Modeling Human-in-the-Loop Security Analysis and Decision-Making Processes	**5. FUNDING NUMBERS**
6. AUTHOR: CDR Michael Schumann	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ______N/A________.

12a. DISTRIBUTION / AVAILABILITY STATEMENT: Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words):

Processes are a fundamental component of most activities undertaken by humans. In software engineering and information assurance, in particular, it is important that processes be understandable, documented, and repeatable so as to ensure that the process outcomes are consistent and predictable. This dissertation provides a novel approach to process creation, documentation, checking, and maintenance that applies mathematical formalism to the engineering of processes that rely in large measure on human decision-making to advance the process flow. However, the modeling approach is sufficiently general for application to any process. This dissertation advances the state-of-the-art in software engineering by providing a formal computer-assisted end-to-end way to conduct requirements engineering. This dissertation advances the state-of-the-art in information assurance by developing a systematic approach that makes the creation of security processes precise and uses formal methods to allow upfront validation and runtime verification of modeled processes. This dissertation demonstrates the modeling approach through a case study of the Unified Cross Domain Management Office's Cross Domain Solution Workflow process.

14. SUBJECT TERMS: Software Engineering, Information Assurance, Process Modeling, Statechart Assertions, Formal Methods, Certification and Accreditation	**15. NUMBER OF PAGES** 160
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

USE OF STATECHART ASSERTIONS FOR MODELING HUMAN-IN-THE-LOOP SECURITY ANALYSIS AND DECISION-MAKING PROCESSES

Michael A. Schumann
Commander, United States Navy
B.A., Washington State University, 1991
M.S., Naval Postgraduate School, 2002

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2012

Author: ______________________________
Michael A. Schumann

Approved by:

James Bret Michael Professor of Computer Science Dissertation Supervisor	Dan C. Boger Professor and Chair of Information Sciences
George Dinolt Professor of Practice	Doron Drusinsky Associate Professor of Computer Science
Duminda Wijesekera Professor of Computer Science, George Mason University	

Approved by: ______________________________
Peter J. Denning, Professor & Chair, Department of Computer Science

Approved by: ______________________________
Douglas Moses, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Processes are a fundamental component of most activities undertaken by humans. In software engineering and information assurance, in particular, it is important that processes be understandable, documented, and repeatable so as to ensure that the process outcomes are consistent and predictable. This dissertation provides a novel approach to process creation, documentation, checking, and maintenance that applies mathematical formalism to the engineering of processes that rely in large measure on human decision-making to advance the process flow. However, the modeling approach is sufficiently general for application to any process. This dissertation advances the state-of-the-art in software engineering by providing a formal computer-assisted end-to-end way to conduct requirements engineering. This dissertation advances the state-of-the-art in information assurance by developing a systematic approach that makes the creation of security processes precise and uses formal methods to allow upfront validation and runtime verification of modeled processes. This dissertation demonstrates the modeling approach through a case study of the Unified Cross Domain Management Office's Cross Domain Solution Workflow process.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BPM	Business Process Model
BPMN	Business Process Modeling Notation
C&A	Certification and Accreditation
CDIP	Cross Domain Implementation Process
CDS	Cross domain Solution
CNSS	Committee on National Security Systems
CNSSI	Committee on National Security Systems Instruction
DAL	Data Abstraction Layer
DCI	Director of Central Intelligence
DCID	Director of Central Intelligence Directive
DIL	Data Interface Layer
DNI	Director of National Intelligence
DOD	Department of Defense
DODI	Department of Defense Instruction
DSM	Design Structure Matrix
HCP	Human Collaboration Processes
HP/A	Human Processes and Artifact
HSPD	Homeland Security Presidential Directive
IA	Information Assurance
ICD	Intelligence Community Directive
IDE	Integrated Development Environment
MDA	Maritime Domain Awareness
MLAS	Multi-Layer Access Solution
MLS	Multi-Level Security
MSPCC	Maritime Security Policy Coordinating Committee
NGO	Non-Governmental Organization
NPD	New Product Development
NPS	Naval Postgraduate School
NSMS	National Strategy for Maritime Security

NSPD	National Security Presidential Directive
ODNI	Office of the Director of National Intelligence
PRC	Peoples Republic of China
R & D	Research and Development
SA	Situational Awareness
SOA	Service Oriented Architecture
SLANG	SPADE Language
SPADE	Software Process Analysis, Design, Enactment
TDSS	Trusted Data Sharing Solution©
TOS	Trusted Operating System
TPM	Time-out Point Manager
TSA	Traditional Structured Analysis
UCDMO	Unified Cross Domain Management Office
UN	United Nations
V&V	Validation and Verification
WFM	Workflow Management
WMD	Weapons of Mass Destruction
WTG	White-box Test Generator
YAWL	Yet Another Workflow Language

ACKNOWLEDGMENTS

This project would not have been possible without the support and assistance of numerous individuals. I sincerely appreciate the help, guidance, and counseling toward this accomplishment.

First, I would like to thank CAPT Kathryn M. Hobbs, USN, Ret. and CAPT Janice Wynn, USN, Ret. both of who made it possible for me to continue in the Software Engineering doctoral program and extend my graduation date in order to accommodate a transition to sea duty.

I would like to thank VADM David Buss and RDML Terry Kraft for the many discussions on operational art, deep thinking, command and control in a denied/degraded environment, and Information Operations; all of which fomented security related thought that improved the dissertation.

I would like to thank CAPT Steve Parode, my current commanding officer for his support to complete my dissertation and the funding to support dissertation related travel.

Next, I would like to acknowledge the members of my dissertation committee: Professors Michael, Boger, Dinolt, Drusinsky, and Wijesekara. They tirelessly gave of their time and energy to guide my research efforts, clarify my understanding of complex materials, and provide suggestions that substantively improved the dissertation. Additionally, thank you to Professor Man-Tak Shing who, even though he was not on my committee read and provided valuable feedback on my dissertation. Professor Michael, as my dissertation advisor and supervisor and before that my Master's thesis advisor in 2002, provided the unwavering support, mentorship, and guidance that I needed in order to complete the research.

Finally, I wish to thank my girlfriend Anna and my children, Ashley and Nicholas for their patience, love, and support throughout this long journey.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. STATEMENT OF THE PROBLEM

Processes are a fundamental component of most activities undertaken by humans. According to Webster's New Universal Unabridged Dictionary, a process is "a particular method of doing something, generally involving a number of steps or operations" (Webster and Mckechnie 1983). In software engineering and information assurance, in particular, it is important that processes be understandable, documented, and repeatable so as to ensure that the process outcomes are consistent and predictable. The ability to formally represent and reason about human-based decision-making processes is a prerequisite for implementing these processes in information systems.

Our research presents a novel approach to process creation, documentation, checking, and maintenance. Our statechart assertion-based approach applies mathematical formalism to the engineering of processes. We focus on human-based processes, that is, processes that rely in large measure on human decision-making to advance the process flow; however, this modeling approach is sufficiently general for application to any process.

Our approach utilizes statechart-based formal process modeling as well as the use of embedded statechart assertions to ensure that modeled process adheres to stated requirements, thus providing traceability[1] between the process requirements and the process implementation. The formal nature of our approach can also help the process engineer to reason about the process. We apply formal methods-based tools and techniques in our approach. As Monin points out, formal methods provide us with a precise and unambiguous means of specifying and reasoning about the behavior of systems (Monin and Hinchey 2003). Formal methods are most frequently used in the software engineering of highly automated security- and safety-critical systems. However, our research demonstrates the use of formal methods to specify and reason about

[1] We apply the IEEE 610.12-1990 definition of traceability stated as, "The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement it satisfies."

primarily human-based processes, such as the process used by the U.S. government to implement, certify, and accredit cross domain solutions (CDS). This process is titled the CDS Workflow and serves as a demonstrative exemplar of our modeling approach.

The intent of our approach is to impart a high degree of precision to our understanding of the process, as well as to provide an automated means of validating that the process does what we expect it to do. In addition, our approach provides for runtime monitoring of the process in execution. Runtime monitoring is useful for both validation which is about answering the question, “Is our formal specification of the natural language description of the process correct?” as well as verification which is about answering the question, “Have we correctly implemented the process?” Runtime monitoring is possible because one of the artifacts produced by our modeling approach is an executable representation of the process. In essence, runtime monitoring uses an executable version of the process and assertions about the process to evaluate input scenarios and classify them as good or bad (Drusinsky 2006).

We apply a particular technical solution, the TimeRover statechart-based modeling tool, to our exemplar process in order to demonstrate the technical feasibility of the approach.

Not all processes require a fine-grain level of modeling, high level of fidelity, or formal specification. For example, Netflix, Incorporated describes a process of rapid recovery from problems that directs employees to “just fix problems quickly” (Anon.). The philosophy behind this is that the company resides in a creative-inventive market where fixing problems is cheaper than preventing them vice a safety- or security-critical market where preventing problems is cheaper than fixing them (Hall 2005). The company goes on to describe the difference between “good” and “bad” processes. The “good” processes tend to be loosely defined (e.g., website push every two weeks rather than random) and would likely not benefit from the use of formal methods.

In a number of arenas there is a need to seamlessly share and integrate information from multiple security domains via a ubiquitous sharing and arbitration mechanism—in systems that perform this function are defined as cross domain solutions

(Committee on National Security Systems 2006). At the same time, we must have the evidence necessary to ensure the prevention of inadvertent disclosure of sensitive or classified information. Prior to September 2008 the U.S. government had an established process for the certification and accreditation[2] (C&A) of high-assurance systems as delineated in Director of Central Intelligence Directive (DCID) 6/3 Policy, "Protecting Sensitive Compartmented Information within Information Systems" (Committee on National Security Systems 2006). The entire C&A process was reviewed and revised as delineated in the Intelligence Community Directive (ICD) 503, "Information Technology Systems Security Risk Management, Certification and Accreditation" and the associated documents within the ICD 503 Framework (see Figure 1) (Director of National Intelligence 2008).

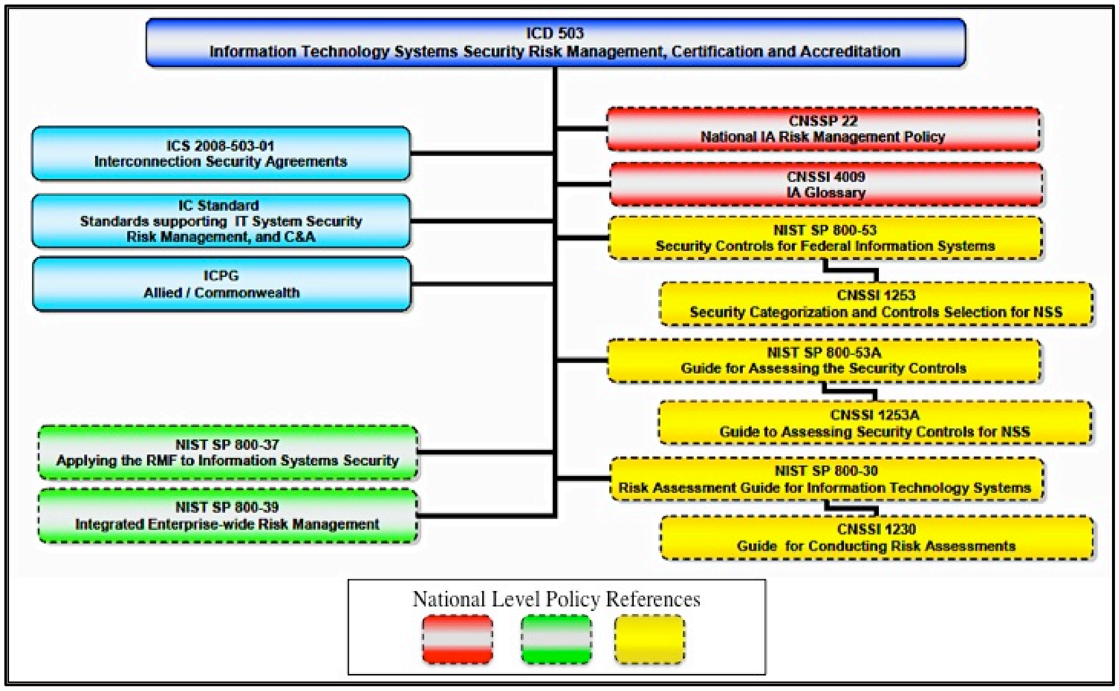

ICD 503
Information Technology Systems Security Risk Management, Certification and Accreditation
ICS 2008-503-01
Interconnection Security Agreements
IC Standard
Standards supporting IT System Security Risk Management, and C&A
ICPG
Allied / Commonwealth
NIST SP 800-37
Applying the RMF to Information Systems Security
NIST SP 800-39
Integrated Enterprise-wide Risk Management
CNSSP 22
National IA Risk Management Policy
CNSSI 4009
IA Glossary
NIST SP 800-53
Security Controls for Federal Information Systems
CNSSI 1253
Security Categorization and Controls Selection for NSS
NIST SP 800-53A
Guide for Assessing the Security Controls
CNSSI 1253A
Guide to Assessing Security Controls for NSS
NIST SP 800-30
Risk Assessment Guide for Information Technology Systems
CNSSI 1230
Guide for Conducting Risk Assessments
National Level Policy References

Figure 1. ICD 503 Framework

[2] Where applicable, definitions for information assurance related terms will be as specified in the Committee on National Security Systems Instruction No. 4009, National Information Assurance Glossary. Accordingly, certification refers to the comprehensive evaluation of security safeguards to support the accreditation process. Accreditation is the formal declaration that an information system is approved by a designated authority to operate at an acceptable level of risk.

The ICD 503 C&A process is embedded within the CDS Workflow process described above. We must be able to understand and ensure the rigorous application of the ICD 503 C&A process for high-assurance systems. In doing so, we help build the evidence necessary for operating these systems at the highest levels of assurance. Currently, there is no rigorously defined mathematical model of the C&A process. Our modeling approach provides a means of building such a mathematically rigorous model. The ICD 503 C&A process is the product of a series of transitional working groups to reformulate and unify the U.S. Department of Defense (DoD) and Intelligence Community (IC) C&A processes. The intent was to develop a single, federated process applicable to high-assurance information systems throughout the federal government. These working groups primarily consisted of domain experts in fields related to high-assurance systems, such as: system certifiers and accreditors, high-assurance system vendors, and DOD and IC chief information officers. The working groups used a combination of domain knowledge, tribal knowledge, best practices, and input from government information assurance (IA) and C&A communities, to develop the process (Niles 2002). While this method can be an effective way of developing processes, in the domain of high-assurance system certification and accreditation, it is not enough. As Gabbar pointed out, formal representation provides a systematic framework to construct and validate the syntax of the underlying system towards building standard representation approaches (Gabbar 2006, 23). Michael et al. show us that (see Figure 2) we can translate customer requirements to formal specification then employ validation and verification (V&V) throughout the development process in order to ensure that the model satisfies stakeholder expectations (Michael et al. 2011).

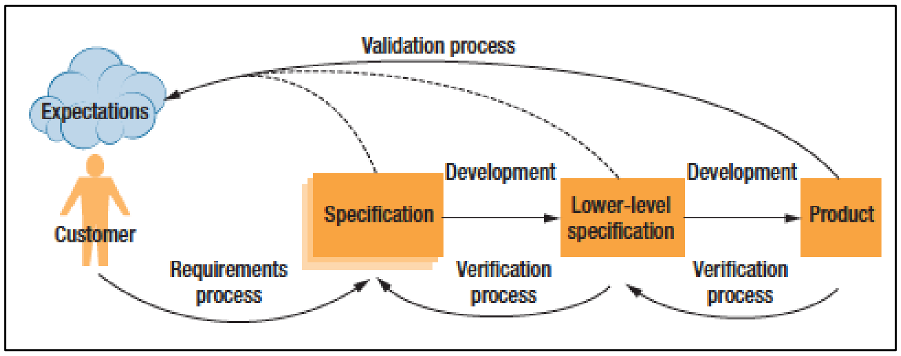


Figure 2. A Continuous V&V Process (From Michael et al. 2011)

We have analyzed the CDS Workflow process in terms of our modeling approach and used the approach to develop a formal process model. We do so using formal methods tools and techniques such as those described in (Drusinsky 2006; Gabbar 2006; Monin and Hinchey 2003). The resultant model provides the level of formality necessary for rigorously applying the exemplar process.

B. SIGNIFICANCE OF THE PROBLEM

The significance of the problem is clear. In order to understand fully a series of activities conducing to an end, especially when that end directly relates to the level of trust we place in high assurance systems, we must be able to rigorously articulate the process, consistently predict the process outcomes, and enforces requirements on the process. In this section, we discuss the importance of formally modeling the CDS Workflow exemplar process as a means of demonstrating the significance of the problem.

In order to implement, certify, and accredit systems for operation at the highest levels of assurance, we need to be able to both understand and trust in the process through which we implement, certify, and accredit those systems. Ultimately, our work in defining a formal model of the CDS Workflow process helps build the evidence necessary to certify and accredit high-assurance CDS systems. However, a well-defined, validated, and documented process is only enough to guarantee safety. The reification (i.e., mapping) from the formal model to the implementation is also needed to guarantee security properties. In other words, validation and verification (V&V) of the process is a necessary but not sufficient condition for obtaining a trusted system.

The Maritime Domain Awareness (MDA) research group at the Naval Postgraduate School (NPS) has partnered with numerous defense research and development (R & D) organizations to develop a prototype system (see Figure 3) called Radiant Alloy that is capable of fusing data from multiple security domains into a comprehensive picture of the Maritime Domain.

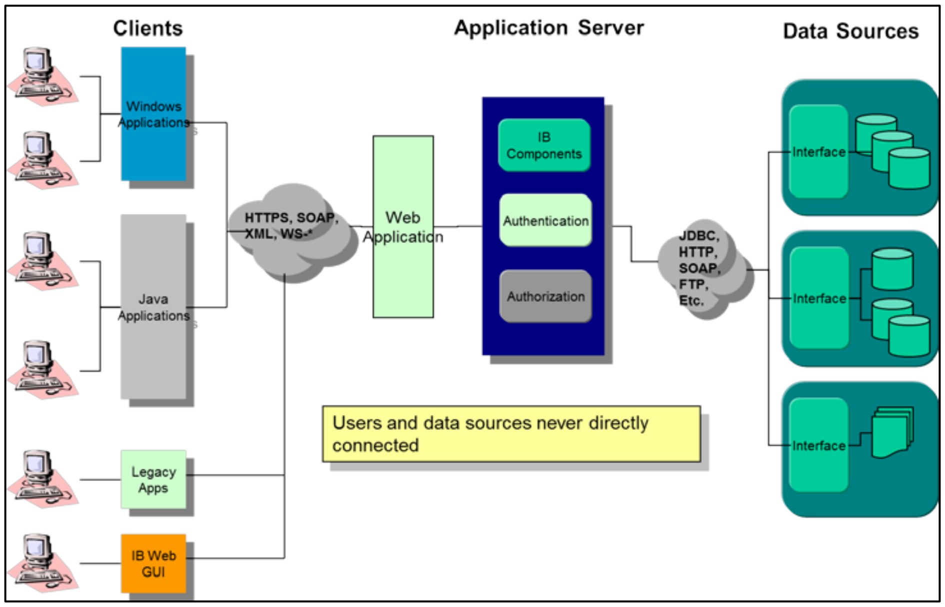


Figure 3.   Proposed System Architecture

In the prototype system, data sources and clients may reside within any of the possible security domains (e.g., UNCLASS, SECRET, TOP SECRET). Therefore, in order to prevent the inadvertent disclosure of classified data (i.e., information leakage), the application server depicted in Figure 3 must prevent the unintended transfer of data between security domains. At the same time, it must allow authorized down/upgrading and transference of data in order to supply those on the client-side with the most comprehensive MDA situational awareness (SA) picture available based on the client's level of access as demonstrated in Figure 4 below. In addition, there must be a means for providing anonymity to the suppliers of the data contained in the repositories.

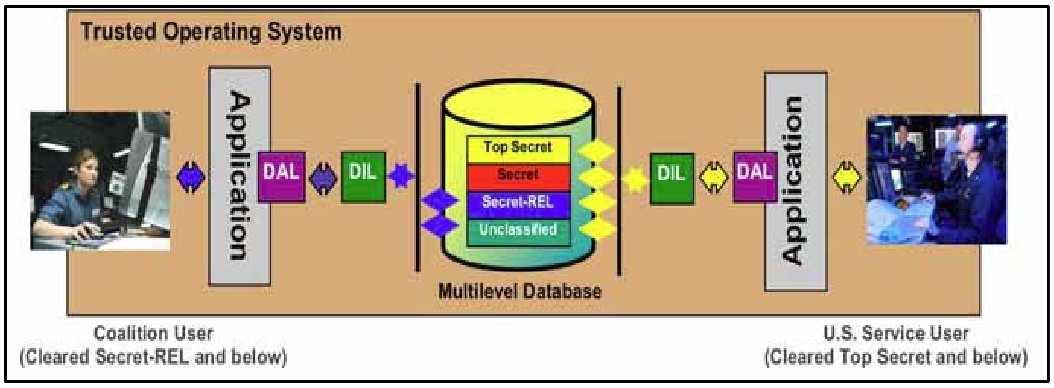


Figure 4. Access Restricted Based on Authorized Access Level of User

Systems such as Radiant Alloy are critical to the future of MDA. The United States and its allies and other groups requiring access to MDA information at a variety of levels need a fully developed, easily accessible, comprehensive picture of the maritime domain. For example: the struggle against military and terrorist forces bent on attacking and destroying U.S. and allied forces, combating the illicit worldwide movement of human cargo (e.g., the slave trade and illegal immigration), and investigation and prevention of narcotics trafficking. In order to realize this vision, information from all security domains must be accessible on an as-needed basis to those that require it and only within authorized security domains. For example, in a scenario involving tracking and interdiction of a cargo ship suspected of carrying concealed weapons of mass destruction (WMD) or WMD components, numerous organizations participate in the effort. These roles have different security levels and data requirements; yet, in order to work together they must be able to share data. This effort would include numerous participants such as: port security guards enforcing entry point access controls, Coast Guard harbor patrol personnel in a law enforcement capacity, watch officers at one of the Coast Guard Regional Fusion Centers that deconflict and manage inbound shipping, ship-refueling operators that routinely collect and analyze data on inbound shipping fuel levels, maritime patrol aircraft that contribute to maritime situational awareness, Naval carrier strike groups which routinely analyze the operating patterns of underway vessels, the U.S. State Department which understands normalized flow of trade between countries, non-governmental organizations (NGOs) that may have unique insights and access to activity at ports of debarkation worldwide, and any other organization that may be able to contribute to the tracking and interdiction of the target vessel.

The governance of high-assurance systems C&A has transitioned to a series of publications that fall within a proposed framework for ICD 503 shown in Figure 1. ICD 503 is a result of a shift in responsibility for the C&A of high-assurance systems from the Director of Central Intelligence (DCI) to the Director of National Intelligence (DNI). The Unified Cross Domain Management Office (UCDMO), under the auspices of the DNI, is responsible for creating, publishing, and maintaining ICD 503. The intent of ICD 503 is to combine current paths to C&A into a unified federal government-wide process. This is a major shift from today's C&A environment where the process of C&A is dependent on the domain in which a system operates. Furthermore, C&A via one system does not transfer or correlate to C&A via another. For example, the Department of Defense uses DoD Instruction (DODI) 8500.2 to govern the C&A of information systems whereas the Intelligence Community (IC) uses DCID 6/3. A cross domain solution fully certified under DODI 8500.2 would still need to go through the DCID 6/3 C&A process in order to be certified for use within the IC. Under the rubric of ICD 503, a single C&A process will be used for all federal information systems.

The ICD 503 framework establishes the authorities and structure for protecting national intelligence information and information systems, whether classified or unclassified (Director of National Intelligence 2008). The Committee on National Security Systems Instruction (CNSSI) 4009 provides a common lexicon for discussing information assurance and national security systems (Committee on National Security Systems 2006).

CNSSI 1199, a product of the U.S. Committee on National Security Systems, provides a means of categorizing U.S. national security systems in terms of the potential impact of unauthorized disclosure of the information residing on the system. As shown in Table 1, this categorization is broken down into three security objectives: Confidentiality, Integrity, and Availability. Each of these is assessed in terms of information related to that security objective and the potential impact of its unauthorized disclosure.

Table 1. Potential Impact Definitions for Security Objectives (From National Institute of Standards and Technology 2004)

	Potential Impact		
Security Objective	**Low**	**Moderate**	**High**
Confidentiality Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. [44 U.S.C. SEC. § 3542]	The unauthorized disclosure of information could be expected to have a limited adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.	The unauthorized disclosure of information could be expected to have a serious adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.	The unauthorized disclosure of information could be expected to have a severe or catastrophic adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.
Integrity Guarding against improper information modification or destruction and includes ensuring information non-repudiation and authenticity. [44 U.S.C.,§ SEC. 3542]	The unauthorized modification or destruction of information could be expected to have a limited adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.	The unauthorized modification or destruction of information could be expected to have a serious adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.	The unauthorized modification or destruction of information could be expected to have a severe or catastrophic adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.
Availability Ensuring timely and reliable access to and use of information. [44 U.S.C., § SEC. 3542]	The disruption of access to or use of information or an information system could be expected to have a limited adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.	The disruption of access to or use of information or an information system could be expected to have a serious adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.	The disruption of access to or use of information or an information system could be expected to have a severe or catastrophic adverse effect on organizational operations, organizational assets, individuals, other organizations, or the national security interests of the United States.

A security category is based on the potential impact of unauthorized disclosure of information residing on the system. The security category is determined by the following equation:

> **Security Category** of information type = {(**Confidentiality**, impact), (**Integrity,** impact), (**Availability**, impact)}, where the acceptable values for potential impact are LOW, MODERATE, or HIGH as described in Table 1 (National Institute of Standards and Technology 2004).

If a U.S. system contains information at different impact levels, the highest impact level is used when determining the security category. The U.S. government has adopted security labels based on a set of definitions for security objectives. These are distinct from internationally developed and recognized definitions such as those contained in the Common Criteria for Information Technology Security Evaluation. For example, the CDS described earlier in this document is designed to interface with security domains from UNCLASSIFIED all the way up to TOP SECRET as shown in Figure 4. Traditionally, information labeled TOP SECRET and above is given the highest levels of protection as unauthorized disclosure of this information has a HIGH potential impact on national security. Therefore, the security category of this system would be (Confidentiality, H), (Integrity, H), (Availability, H). Within the ICD 503 Framework this would be referred to as an ICD 503 H-H-H security category.

The successful culmination of our research provides a formalized, statechart-based approach to human-based process modeling. We demonstrate this approach using the largely human-based Cross Domain Implementation Process (CDIP) and CDS Workflow process as exemplars. Through formal methods tools and techniques, we show that it is possible to rigorously define, monitor in execution, maintain, and enforce requirements on these processes. A benefit of this approach is continuous monitoring of the process if we implement a runtime model of the process with assertions. We further show that our modeling approach is generalizable to any process.

C. RESEARCH HYPOTHESIS AND APPROACH

1. Research Hypothesis

We can extend the use of statechart assertions through the application of a process modeling approach, where:

- The approach provides a systematic, formal methods-based procedure for precise development, debugging, runtime monitoring, long-term maintenance, and upfront validation and verification of decision-making processes.
- Integrated statechart assertions serve as a requirements enforcement mechanism on the modeled process.

We test our hypothesis vis-à-vis the application of the modeling approach to the CDIP and its successor the CDS Workflow process.

2. Research Approach

Using formal methods tools and techniques, we develop a systematic approach to process-modeling engineering. This approach provides us with a systematic, repeatable basis for engineering, understanding, and assessing the security properties of processes developed with the modeling approach. Formal models are a useful tool for helping us understand and clearly describe a system or process in unambiguous language. Our modeling approach allows assertions to be exercised and visualized via animation. This can be used as a communication tool for discussing the process and system with decision makers, process owners, management, and other stakeholders.

The nature of the largely human-based CDIP and CDS Workflow process is such that it involves subjective and objective sub-processes. By subjective, we mean processes that involve elements of human evaluation. For example, evaluating the effectiveness of an authentication mechanism is a subjective process where the evaluator's experience, knowledge, personal bias, and definition of the word "effectiveness" can affect the process outcome. By objective, we mean processes that are repeatable, measurable, well structured, and produce predictable outputs for a given set of inputs. For example, evaluating the strength of passwords is an objective process definable in terms of the computing power and time necessary to crack the passwords successfully. Figure 5 represents the CDS Workflow process as a composition of these subjective and objective sub-processes. As part of our research, we break the CDIP and CDS Workflow down into its subjective and objective component parts. We then use formal methods techniques to model the objective sub-processes in a well-defined formal language.

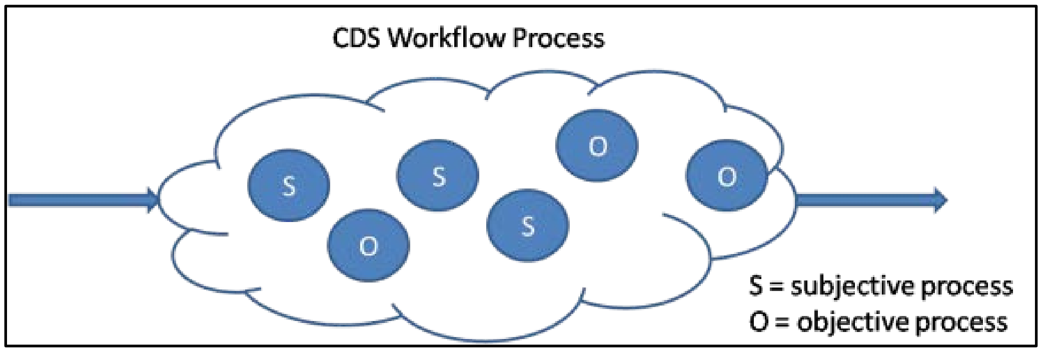


Figure 5. Conceptual View of CDS Workflow Process

As Monin points out, when a formal model is available, we can state with precision the properties we expect from the system (i.e., the system's behavior) and then formally verify them (Monin and Hinchey 2003, 4). Our approach to modeling provides for precisely stating the properties we expect from the objective components of the CDS Workflow process. We assess the technical feasibility of our modeling approach in terms of its real-world applicability and provide us with a test case on a process being developed for managing and implementing CDS.

D. CONTRIBUTIONS OF THIS RESEARCH

1. Software Engineering

We contributed to software engineering by introducing a novel way for software to automate a new domain, that being process-modeling engineering of high-level, human-based processes.

2. Process-modeling Engineering

We developed a systematic approach to formally modeling, validating and verifying high-level human-based processes. Modeling human-based processes can be challenging to model because of the hard-to-capture elements such as human decision-making, sequencing, and concurrent activities. We applied some of the tools and techniques from software engineering to provide an end-to-end means of modeling, validating, and verifying these processes within the same formalism.

3. Case Studies

We demonstrated the application of our systematic modeling approach through two case studies. These two cases represent hard process modeling problems and encompass a large portion of the hard-to-capture elements mentioned above.

4. Real-world Impact

We provided feedback to the UCDMO on process errors discovered through the case studies, resulting in corresponding changes to the real-world process for requesting, developing, implementing, certifying and accrediting cross domain solutions.

E. THESIS ORGANIZATION

In Chapter II we assess and discuss relevant literature. In Chapters III and IV we describe and exposit the modeling approach and provide usage examples. In particular, Chapter III provides a detailed description of the modeling approach and its applications while Chapter IV applies the modeling approach to two particular processes. The final chapter (Chapter V) provides conclusions and a view into future research.

II. RELATED RESEARCH

A. INTRODUCTION

There is a long history of research on the modeling of software and security processes, such as those for development, maintenance, and in particular for V&V and C&A of software and information systems. For example, the First International Conference on the Software Process held in Redondo Beach, CA in 1991. At this conference, many papers were presented on the topic of formally modeling software engineering processes. This conference was primarily focused on high-level processes such as those by which software is developed as opposed to low-level processes such as those managed by an operating system (e.g., memory or file management). Subsequent to the conference, the field of formal process modeling continued to evolve not only for software engineering processes but also for business processes. Researchers in both fields experimented with a wide variety of tools and techniques in the effort to find ways to formally specify, validate, and verify high-level software development and business processes.

Additionally, a significant body of research exists on the development of executable process models from formal specifications. However, a gap exists in the open literature with regard to conducting runtime verification of the executable process models. Our work addresses this need by providing a systematic process modeling approach that includes runtime verification of the executable process model via embedded statechart assertions.

B. FORMAL METHODS IN PROCESS MODELING

Gruhn and Emmerich introduced a software process modeling language called FUNSOFT nets. These are essentially Petri nets with a formally defined semantics in terms of Predicate/Transition (Pr/T) nets and extended by multi-sets (Emmerich and Gruhn 1991). Gruhn describes software process modeling as focusing on software process models that can be used for governing software processes with the intent of automatically detecting incongruities between a software process and its associated model. Executable software process models, according to Gruhn, contribute to increased software development productivity and software quality. The underlying Pr/T net contains a multiplicity of features that facilitate the specification and governance of the modeled software processes. These features include the following: a set of jobs representing software development activity; a set of object type definitions that can be attached to channels via a specified function in order to specify the allowable type for a particular channel; a set of predicates to define conditions on the objects; and an initial marking that respects the typing of channels. Gruhn extended the discussion of FUNSOFT nets to encompass social processes represented by software development teams (V. Gruhn 1992). He points out that the incorporation of human social interactions into software process models represents an area of research that is not well understood though he does suggest that the dialogue artifacts inherit in FUNSOFT nets may provide a means of describing these interactions in the context of software processes (Emmerich and Gruhn 1991). Gruhn concludes that the non-linear factors involved in human social interactions prevent the modeler from doing more than pointing out explicitly where human interaction and cooperation impacts software development.

Hibdon and Hartrum examine the development of an organizational process model based on object-oriented design concepts and formal software engineering methods. They apply a sequential design process that builds an informal Rumbaugh model, translates it to a *Z* based specification, and finally translates the *Z* specification into the *Refine* language and builds an executable model via the Software Refinery Environment (Hibdon and Hartrum 1996). The translation of *Z* to *Refine* requires special attention with regard to *Z* predicate constraints since *Refine* does not support Predicate constraints that must always hold true. Constraints of this type must be mapped into pre- and post-conditions of *Refine* functions. In our modeling approach, the statechart-based model and the embedded assertions are designed with the same modeling techniques in the same formalized language.

An experimental application of process modeling technology at the British Airways showed that there is value in using flexible modeling tools as the modeling process can reveal flaws and inconsistencies in the original process. If this results in a change to the original process, the model needs to be changeable to reflect the adjustments (Emmerich et al. 1996). Our modeling approach addresses this issue by using a modeling tool that is flexible enough to apply rapid changes yet continues to ensure that the model maintains consistency with the underlying formal semantic.

Business process models have become increasingly complex, making it steadily more difficult to implement the models within an information system. Koehler et al. suggest a dichotomy exists between the tools and methods used to describe a business process and the tools and methods used to describe the information technology (IT) artifacts implementing the process. They make the case that process requirements should be made explicit and demonstrate the use of basic model checking techniques to verify a model's global properties of *reachability* and *liveness*. These terms are defined as follows: the *reachability* property states a particular situation *can* sometimes be reached whereas a *liveness* property expresses that, under certain conditions, a situation will *ultimately* occur (Koehler, Tirenni, and Kumaran 2002). Here, the term "global properties" refers to those properties that apply to the entire model. While verification of these properties is useful in terms of a basic understanding of how the model behaves during enactment, they do not provide insight or enforcement for properties within the context of the model. In other words, these properties are agnostic to the contents of the model. Our research addresses the need for contextual verification of a model's internal properties through the runtime application of embedded statechart assertions.

Van Dongen, Van Der Alst, and Verbeek as well as Van Dongen and Jansen-Vullers show that process-aware information systems are used to support a wide range of business processes. Often, these systems are configured based on a process model which drives the need to ensure that the process model is correct. Therefore, many researchers have investigated the verification of process definitions with a focus on the construction of mathematically sound and executable syntax and semantics of specific modeling languages. In spite of the importance of having a correct process model the authors indicate that for many process-modeling techniques, mathematically well-defined syntax and semantics do not exist or they are too complex for process designers. In order to demonstrate the value of modeling techniques based on a well-defined language, the authors use the Event-driven Process Chains (EPC) modeling language to describe their approach for verification of EPCs. Then, through a series of reductions, they translate the EPC to a form of the classic Petri net model known as Place/Transition nets which consist of two modeling elements. Their stated goal is to provide the process designer with a tool to find possible problems in a process specification. To that end, they require the process designer to interactively evaluate the EPC and Place/Transition model at two different points in the verification process in order to make decisions about the behaviors exhibited by the model. (Van Dongen, Van Der Aalst, and Verbeek 2005; Van Dongen and Jansen-Vullers 2005) This approach leads to what the authors describe as a relaxed definition of correctness that focuses on giving the process designer the ability to determine whether, according to his personal standards for the process resulting in desirable versus undesirable behaviors, a process under examination is *correct*.

As pointed out by Gruhn and Lane, the building of business process models (BPM) can benefit from well-established practices in software engineering (V. Gruhn and Laue 2007). The focus and main contribution of their research is a discussion of the value of style checking in improving the quality of BPM. The authors suggest that significantly improving the quality of BPMs related to software development using style rules and style checking leads to an improvement in the quality and success of enterprise software development.

Business Process Modeling Notation (BPMN) is an emerging standard that allows business processes to be captured in a standardized format. BPMN lacks formal semantics which leaves many of its features open to interpretation and hinders verification of processes described in BPMN. Ye et al. proposed a methodology for mapping a subset of BPMN elements to the Yet Another Workflow Language (YAWL) specification language formal, set-notation based definitions. (Ye et al. 2008) The authors developed a tool for automated translation of a BPMN model to a YAWL net. While this tool required preprocessing and did encounter translation errors it provided an initial step toward the type of formalization required for BPMN model verification. However, it also highlights the challenges of taking a model developed in a language lacking formal semantics and translating that model to a sufficiently formal language to allow verification. Our modeling approach addresses this gap by allowing the process engineer to both design and verify process models within the same formal specification language.

Grady offered a universal architecture description framework (UADF) and showed how this combination of UML and SysML can be applied to modern day problem spaces to provide organized methods for identification of specialty engineering/quality, environmental requirements, product entities, and the map between models and product entities borrowed from Traditional Structured Analysis (TSA) (Grady 2009). He points out the difficulty in connecting a design with the verification process through which we prove that a design satisfies its driving requirements. Our work in applying statechart-based assertions to the modeling of processes addresses this by modeling requirements in the same statechart-based notation as the modeled process and embedding those assertions within the modeled process so that they can be enforced at runtime.

The integration of human interactions into process modeling can be challenging due to the unpredictable nature of human behavior. It is important to find ways to formally specify human interactions within a process in order to facilitate process validation and verification. Zongyang, Liyang, and Hongli propose a formalization of human interactions within business processes through the introduction of the Human Processes and Artifact (HP/A) model. This model applies rigorous, set notation-based definitions to human processes combined with statechart visual representations of the human interactions within a business process. However, the authors identify a gap in the verification of models that incorporate human interactions due to ability of humans to make unpredictable choices (Zongyan, Liyang, and Hongli 2010). Our work contributes to closing this gap in human-based model verification by integrating the representation of human choices within the process models designed through our modeling approach. As a result, requirements or constraints on human choices can be modeled and enforced at runtime using statechart embedded assertions within an executable model. This facilitates runtime verification of models that integrate the representation of human choices.

Hurtado Alegria, Bastarrica, and Bergel point out that the software process models can be sophisticated and large. The formal specification of these models demands an enormous effort and once specified, the process engineer lacks tools to evaluate the quality of the process. They demonstrate the Analysis and Visualization for Software Process Assessment (AVISPA) tool for analyzing and identifying errors within software process models as an *a priori* way to measure the quality of the process prior to execution. The authors point out that formal V&V techniques for measuring and testing discrepancies between a model and its execution can only be carried out on a process model that has been implemented, tailored, and enacted (Hurtado Alegría, Bastarrica, and Bergel 2010; Hurtado Alegría, Bastarrica, and Bergel 2011). Our process modeling approach addresses this concern by providing the process engineer with an iterative approach to process design that integrates V&V throughout and includes the development of an executable representation of the process model.

Karniel and Reich identified a gap between the process planning and process implementation communities. They indicate that many new product development (NPD) projects fail. The design structure matrix (DSM) can be used for planning and modeling the process flow of NPD projects. However, DSM lacks the formality necessary to verify correctness of the process model. The authors suggest a relationship between the NPD project failures and inability to verify the DSM model of the project. They propose a complex series of formal rules to translate a DSM model to a workflow net. Workflow nets are a class of Petri nets with the necessary formalisms and tools to conduct process verification. The authors point out that their approach is difficult to implement for more complex DSM models. (Karniel and Reich) Our work addresses this gap by providing a visual modeling language that is accessible to both the process planning and implementation communities yet includes the necessary formality and tools to enable V&V of the modeled process.

Human interactions are an integral component of business processes. However, as Stuit points out, Human Collaboration Processes (HCP) are either ignored or not handled well by current process modeling approaches. He argues that there is a demand for novel modeling tools for the design and modeling of HCPs in organizations. Stuit demonstrates an agent-based, graphical approach to modeling human interactions that serves as a "necessary precursor for their proper analysis and improvement" (Stuit 2011, 3–5). Our research addresses the modeling of human-in-the-loop decision-making through the artifacts of our statechart based process modeling approach. We apply and enforce runtime constraints on decision-making through the use of statechart embedded assertions.

C. SOFTWARE SAFETY

Bishop provides an introduction to the concepts of information leakage and safety in information systems. These terms are used rather than *secure* and *unsecure* because safety refers to the abstract model and security refers to the actual implementation (Bishop 2002, 47–91):

Definition 3-1. When a generic right *r* is added to an element of the access control matrix not already containing *r*, that right is said to be *leaked*.

Definition 3-2. If a system can never leak the right *r*, the system (including the initial state s_0) is called *safe with respect to the right r*. If the system can leak the right *r* (enter an unauthorized state), it is called *unsafe with respect to the right r*.

The access control matrix model is fundamental to both of these concepts. Access control matrices, originally proposed by Lampson were enhanced by Graham and Denning and are applied to modern day systems by Bishop (Lampson 1974; Denning 1971; Graham and Denning 1972; Bishop 2002). An access control matrix views a system in terms of the set of protected entities, contained in the set of objects *O* its active objects, contained in the set of subjects *S*; and rights drawn from the set of rights R in each entry *a*[*s*,*o*] where $s \in S$, $o \in O$, and $a[s,o] \subseteq R$ entity relationships are captured in a matrix *A* where rights drawn from *R* get assigned to each entry *a[s, o]*. The protection states of a system are then represented by the triple (*S*, *O*, *A*). Within this context, leakage occurs when a generic right $r \in R$ is added to an element of the access control matrix not already containing *r*. The set of authorized states for the system are those in which no command c(x_1, …, x_n) can leak *r*.

Safety as described here is critical for CDS. These systems must employ access controls that guarantee safety in order to prevent the inadvertent transfer or disclosure of sensitive or classified information. Yet, we cannot analyze a system or process in terms of its safety guarantees unless we precisely understand it. Our statechart-based approach to formal process modeling provides the level of precision necessary to facilitate an analysis of the model in terms of its safety guarantees.

D. STATECHARTS

Harel introduced statecharts to address a well-recognized problem with regard to the difficulty of specifying and designing large and complex reactive systems where:

> A *reactive system*, in contrast with a *transformational system*, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. Examples include telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software. (Harel 1987)

His seminal work in the field of visual specifications has been studied extensively and utilized in wide variety of subsequent research on the application of statecharts and their successor, UML statecharts.

Dong and Shensheng demonstrate that statecharts can be used to model business workflows by modeling an international travel agency's process for handling customer travel requests. They show that it is possible to represent hierarchal levels of the workflow and transition between levels by leveraging the AND/OR decomposition of statecharts, which provides the ability to, in effect, "zoom in" and "zoom out" of the model (i.e., move between abstract layers). Additionally, they suggest that the well-defined semantics of statecharts allow for the verification of statechart-based workflow models (Dong and Shensheng 2003; Harel 1987, 233–235).

Drusinsky applied UML statecharts to real-world specification and verification in (Drusinsky, Shing, and Demir 2006; Drusinsky 2006; Drusinsky 2008). Though the concepts and techniques introduced by Harel and Drusinsky focus on using statecharts for the specification and development of complex, reactive, hardware and software systems, we show that these same techniques allow us to formally specify and reason about the largely human-based CDS Workflow process. Drusinsky uses a Java based statechart notation (i.e., any Java statement can be written as a statechart action, any Java condition can be written as a statechart transition guard, and any Java method name can be written as a transition event) as a basis for describing reactive systems (Drusinsky 2006; Drusinsky 2011). In other words, this Java based statechart notation is Turing equivalent. The notion of Turing equivalence in our chosen notation is important as this equivalence relationship tells us that the language described by the notation computes precisely the same class of functions as Turing machines. Therefore, the deep body of research on the power of Turing machines applies to this Java based statechart notation. For additional details, authors such as Sipser, Hopcroft, and Kelley provide a more complete discussion of the Turing machines and their range of computable functions (Sipser 1997; Hopcroft, Motwani, and Ullman 2007; Kelley 1995).

Building on earlier work, Drusinsky and Shing extend UML statecharts to include K-statecharts (Drusinsky and Shing 2009). This construct allows the use of knowledge logic formulae; a form of modal logic used for reasoning about multi-agent systems, for modeling multi-agent systems whose behavior depends on knowledge and belief statements made by the system agents. Their model provides inter-visibility amongst the agents. The ability of an agent to view into and act upon the behavior (i.e., states) of another agent allows the development of formalized, executable models for complex multi-agent systems.

Crane and Dingel explore the syntactic and semantic differences between three different statechart formalisms: Classical, UML, and Rhapsody (Crane and Dingel 2007). Their results indicate a lack of standardization between these formalisms. They show that due to subtle semantic and syntactic differences a model that is a well-formed statechart in all three of these formalisms may exhibit different behaviors in each of the separate formalisms. This is not a concern for the research described in this document as we use only one of the formalisms, UML statecharts.

E. REQUIREMENTS

In the field of formal verification of systems or systems-of-systems, we ensure that the behavior of a subject system complies with its formal correctness specification. However, the formal specifications are often based on natural language (NL) requirements specifications. Drusinsky points out NL specifications are often ambiguous and we must be careful when writing formal specifications from NL in order to ensure that the translation is as accurate and precise as possible. Several ongoing research efforts address this problem. Bruegge and Dutoit articulated a UML-based model for requirements elicitation and analysis that demonstrates the capturing of customer requirements, typically in natural language and subsequent translation to formal or semi-formal notation. The transformation to a more formal notation ensures that system developers work from a common understanding of the requirements provided by system stakeholders (Bruegge and Dutoit 2004, 123–166). Drusinsky showed us how to identify NL requirements of interest from UML analysis diagrams (e.g., activity diagrams, message sequence diagrams) (Drusinsky 2008).

It is important to validate formal requirements specifications to ensure they correctly represent the intended behavior. In the case of requirements specifications written as statechart assertions, Drusinsky, Michael, Otani, and Shing introduced a pattern-based methodology for validating them against their NL requirements. This is particularly useful when the assertions are grouped into libraries of reusable formal specification assertions. The underlying concept for this approach is that statechart assertions are often focused on a specific, coherent concern. This suggests a likelihood of ensuring they correctly represent the intended behaviors by testing them against a finite, representative set of validation scenarios. The pattern-based methodology uses representatives groups of tests (i.e., patterns) such as *obvious success, obvious failure, event repetition*, and *multiple time intervals* to ensure that testing includes the type of scenarios often overlooked in the validation process. (Drusinsky et al. 2008; Drusinsky 2011)

Our work addresses validation of requirements for human-based processes by facilitating the clear, visually appealing articulation of requirements in the same notation used to model a process under examination. When articulated in this manner, post-facto analysis and modification of these requirements may be performed in a rigorous and well-structured manner. Our work addresses the requirements verification concern for software engineering related human-based processes by ensuring process adherence to requirements articulated in statechart assertions and embedded within our statechart-based process models.

F. RESEARCH GAPS

This research identified gaps in the wide body of research on process engineering and process validation and verification. A significant body of research exists on the development of executable software development and business process models from formal specifications. However, a gap exists with regard to conducting runtime verification of the executable process models. Our work addresses this need by providing a formal process modeling approach with runtime verification of the executable process model via embedded statechart assertions. We achieve this by treating human-based processes, conceptually, as reactive systems and applying to them formalized tools and techniques.

We examined research that articulates the challenges of taking a model developed in a language lacking a formal semantic and translating that model to a sufficiently formal language to allow verification. Our modeling approach addresses this gap by allowing the process engineer to both design and verify process models within the same formal specification language.

THIS PAGE INTENTIONALLY LEFT BLANK

III. THE MODELING APPROACH

In this Chapter, we describe the approach by which we apply formal methods tools and techniques to the modeling of partially automated, human-based, C&A processes.

A. FORMAL METHODS TOOLS AND TECHNIQUES

1. Desirable Attributes of Formal Methods to Support the Modeling Approach

Analysis of the research discussed in Chapter II revealed knowledge gaps in the field of applying formal methods to high-level processes. We developed a set of desirable attributes for a formal specification language and associate tools to support the modeling of high-level, human-based processes and address the identified knowledge gaps.

The language should have a well-defined syntax and semantics in order to provide the level of formality necessary to unambiguously model and facilitate V&V of the process. The visual representation of the language should be sufficiently understandable as to provide a good communication medium between users, stakeholders, and process engineers. The language needs to provide mechanisms or artifacts that allow the formal specification of human-in-the-loop decision-making. The language needs to be able to represent attributes such as hierarchy and concurrency that are often found in complex human-based processes. The formal specification of the model as well as the artifacts necessary for verification of the model should all be expressible in the same language. Table 2 shows a comparison chart for the languages.

The language's associated tools should provide a mechanism for building a visual representation of the models using the chosen language. The modeling tools should provide the flexibility to make adjustments to the process model as necessary. They should also be able to generate an executable representation of the model. The associated tools should provide the ability to conduct runtime verification of the modeled process, an automated means of verifying an executing model's adherence to specified properties or requirements. This would require a means of monitoring the process model in execution and enforcing desirable runtime properties of the executing process model, such as ensuring adherence to temporal constraints.

We hypothesize that a formal language and associated tools that have, at a minimum, the listed attributes will enable the formal specification, maintenance, validation, and runtime verification of a model that accurately represents a partially automated, human-based, C&A process and provide a viable communication medium for discussing the process among users, stakeholders, and process engineers.

2. Assessment of Formal Methods for Desirable Attributes

We assessed several formal languages and their associated tools, such as Communicating Sequential Processes (CSP) (Hoare 1985), Petri nets (Emmerich and Gruhn 1991), the Z formal language, and UML statecharts (Drusinsky 2006) to determine whether they possessed the attributes listed in Section III.A.1. These languages are compared in Table 2 on the basis of how closely each on matches our set of desirable attributes.

Table 2. Desirable Attributes of a Formal Language for Process Modeling Approach

	Language			
Description of Attributes	**CSP**	**Petri Nets**	**Z**	**UML Statecharts**
Well-defined syntax/semantics	☑	☑	☑	☑
Easily understandable (visual)	☐	☑	☐	☑
Specify human decision-making	☑	☑	☐	☑
Represent nesting	☑	☐	☑	☑
Represent hierarchy	☐	☐	☑	☑
Represent concurrent activities	☑	☑	☑	☑
Model/verify in same language	☑	☐	☑	☑

Ryan and Schneider used CSP as a modeling mechanism for a variety of security protocols (Ryan and Schneider 2000). Wong and Gibbons demonstrated a technique for representing BPMN process models in CSP in order to provide a semantics for formal analysis and comparison of BPMN diagrams (Wong and Gibbons 2008). As discussed in Chapter II of this document, CSP is based on a relatively complex mathematical notation. Many researchers in the field of process modeling have discussed the importance of reducing the complexity and increasing the ease of understanding formal process models in order to better communicate with process users, stakeholders, and designers

(Emmerich et al. 1996; Koehler, Tirenni, and Kumaran 2002; Zongyan, Liyang, and Hongli 2010; Hurtado Alegría, Bastarrica, and Bergel 2010; Hurtado Alegría, Bastarrica, and Bergel 2011; Karniel and Reich).

Many researchers have applied Petri nets to high-level processes as a means of formal specification. The works of Emmerich and Gruhn (1991); Gruhn (1992); Van Der Aalst and Van Hee (2004); and Van Dongen, Van Der Aalst, and Verbeek (2005) demonstrate a variety of methods of applying Petri nets as a tool for process modeling. The graphical nature of Petri nets makes them an excellent tool for communicating about a process under examination. However, researchers have pointed out that Petri nets do not scale well for the visual representation of large, complex processes as the basic Petri net formalism lacks artifacts for representation of hierarchy. Clempner proposed an extension to represent hierarchy in a subclass of Petri nets known as Decision Process Petri Nets (DPPNS) though his work is formative in nature (Clempner 2010).

Hibdon and Hartrum built an executable model of a U.S. Air Force component known as a wing. Their modeling process required creation of an informal Rumbaugh object model (Rumbaugh et al. 1991), translation to the formal language *Z*, and subsequent translation into Refine constructs for execution (Hibdon and Hartrum 1996). The final product of their multi-step approach was an executable model; however, the *Z* language and the Refine construct are both complex, non-visual representational formalisms.

UML statecharts are a visual formalism that has been used for representation and formal specification of systems, architectures, and processes. As discussed in Chapter II of this document, researchers have demonstrated that UML statecharts have well-defined semantics with artifacts expressive enough to capture elements of human-in-the-loop decision-making. They have been demonstrated as an effective visual communication mechanism for communicating about processes (Dong and Shensheng 2003). Specifications written as statechart assertions and embedded within a statechart-based process model enable runtime verification of the model. UML Statechart assertions are a class of statecharts and as such, written in the same language.

UML statecharts possess the desired attributes for a formal specification language that we outlined in Section III.A.1. This formal language was the best fit in terms of its potential for use in addressing the research gaps identified in Chapter II.

After deciding on a language that satisfies the stated desired attributes, we needed to determine if any of the currently available tools for working with UML statecharts would satisfy the requirements stated in Section III.A.1. Both research based and commercially based tools exist for the design and manipulation and statechart-based models. We look at several of these tools such as VisualSTATE, Yakindu, and StateRover (IAR Systems 2012; Muelder 2011; Drusinsky 2006).

VisualSTATE is standalone statechart-based modeling tool that also provides a point-and-click interface for easy development and editing of models. This software has a built-in module for code-generation to automatically create an executable representation of the model in C++. The software automatically performs syntactic verification to ensure model compliance with the underlying language rules. The verification module includes the functionality for static analysis of the model to ensure compliance with both pre-defined and custom properties. VisualSTATE has a dynamic analyses module that can provide an animated view of how specific events affect a model. Events are fed into the simulation via an interface with the ability to replay sequences of logged events. However, VisualSTATE does not include the functionality to enforce requirements in conjunction with runtime execution monitoring.

Yakindu is a statechart-based modeling tool that operates as a plug-in for the Eclipse integrated development environment (IDE). Yakindu is a visual modeling tool that does provide a mechanism for automated generation of an executable model. It has a point-and-click interface that makes it easy to build and dynamically adjust models. The Yakindu plug-in has the capability to interface with an external code-generator module capable of mapping a statechart model to C or Java source code. However, the code generator module is experimental and must be installed separately from the Yakindu plug-in. The plug-in applies automatic syntactic verification rules to each statechart model and reports discrepancies to the model developer via both visual and textual cueing. Yakindu provides a simulation function that executes the generated code but it does not provide the ability to conduct runtime verification.

StateRover is a UML statechart-based modeling plug-in for the Eclipse IDE. This tool includes a built-in code generation module that automatically maps a statechart model to C, C++, or Java source code. Model design is accomplished through a point-and-click interface that makes it easy to build and dynamically adjust or reuse models. The tool includes an automated syntactic and semantic validation module to ensure model compliance with the underlying statechart syntax rules and semantics. In addition, code generation will not run unless the model is able to successfully pass the syntactic and semantic validation with no errors. StateRover provides the functionality for runtime verification of a statechart model through the application of embedded statechart assertions enforced within the model during execution. StateRover provides an integrated, white-box[3] test generator that builds test cases for use in automated testing. The generated test cases are used in within the JUnit[4] test framework to provide runtime execution monitoring of the model as it enacts the generated test cases. Embedded statechart assertions serve to enforce runtime properties or constraints placed on the model (e.g., temporal constraints).

Of the statechart modeling tools surveyed, StateRover possesses the desired attributes outlined in Section III.A.1 for a tool designed to enable modeling in our chosen formal language. This tool was a best-fit in terms of its potential for use in addressing the research gaps identified in Chapter II.

B. PROCEDURE

The diagram shown in Figure 6 outlines our process modeling approach. Solid lines represent the primary procedural flow path. Dashed lines represent ongoing communication with process stakeholders to ensure a modeled process aligns with and

[3] The white-box test generator is discussed in Section III.C.6.

[4] The JUnit test framework is discussed in Section III.D.3.

achieves their desired outcomes as originally specified in stakeholder requirements. This ongoing feedback loop is one component of model validation.

Our approach provides the level of formalism necessary to rigorously specify a partially automated, human-based, C&A process and conduct runtime verification on that process to ensure the process behaves exactly as it was designed. This allows the process engineers and stakeholders to ensure that the process flows exactly as it is intended. Formalization will significantly improve the final product of a process for developing, implementing, and C&A of cross domain solutions designed to facilitate and guard the flow of information between various security domains. Formalizing the process will help ensure that it provides a product that is well-defined, well-developed, and consistent in its execution.

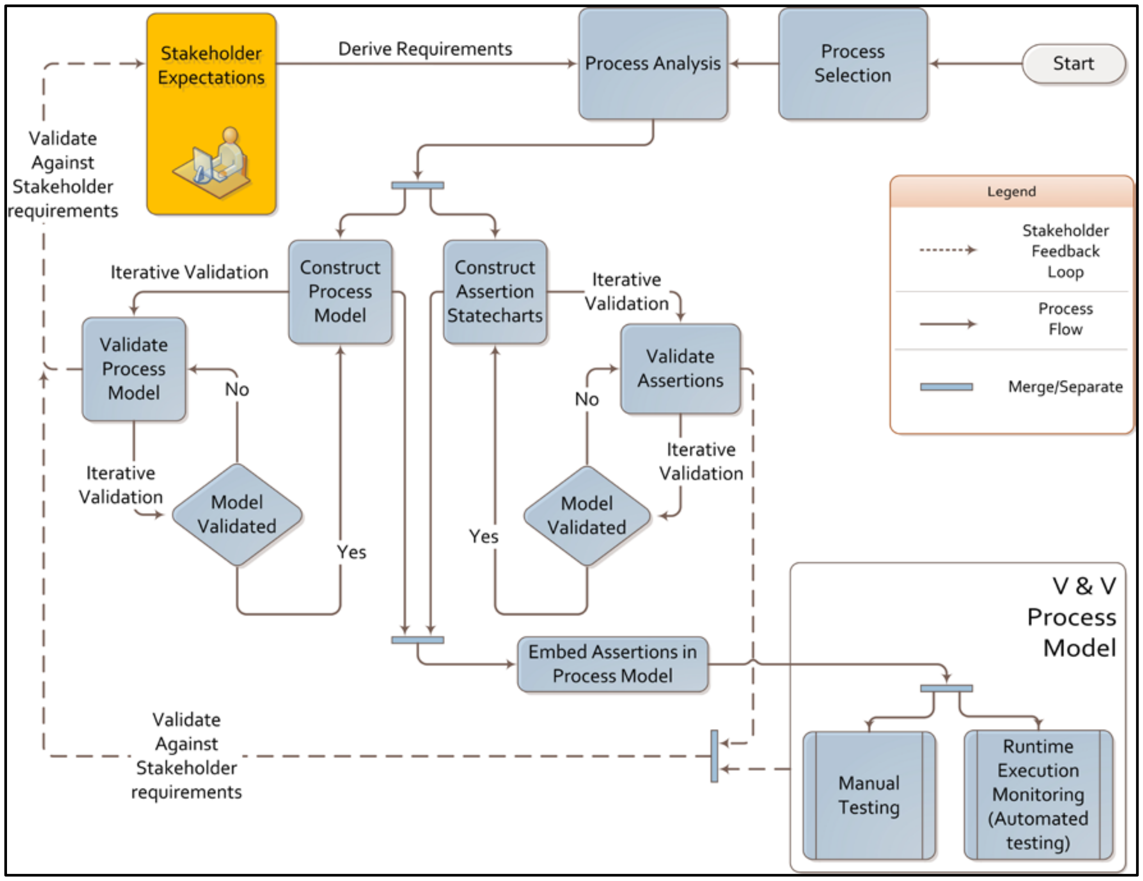


Figure 6. Overview of Statechart-Based Process Modeling Approach

1. Iterative Design

Our modeling approach allows the process engineer to iteratively create a process model in conjunction with creation of the process itself. Once the process is established, the process engineer can use this approach to adjust the model throughout the lifecycle of the underlying process.

Berry and Wing tell us that a second look at the thing being formalized can result in a better product, since for any large or highly complex project, one must understand the problem – a lack of understanding can lead to catastrophic failures. They go on to suggest that such an understanding is more likely to be achieved when building a "complete" model of the intended system (e.g., a formal specification or a prototype) (Berry and Wing 1985). Our approach facilitates such an understanding of the modeled process through construction of the UML statechart-based process model. For example, Figure 7 is a process outline of the CDIP provided by the UCDMO while Figure 8 shows the statechart-based model developed through our modeling approach applied to the CDIP.

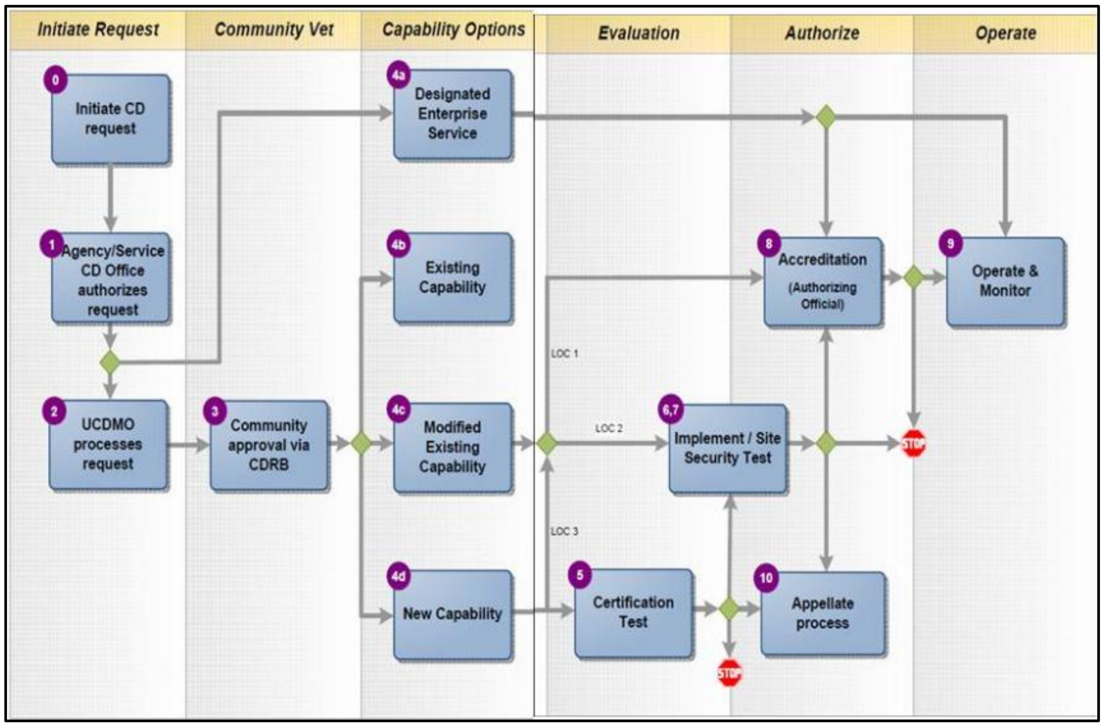

Initiate Request
Community Vet
Capability Options
Evaluation
Authorize
Operate
0 Initiate CD request
1 Agency/Service CD Office authorizes request
2 UCDMO processes request
3 Community approval via CDRB
4a Designated Enterprise Service
4b Existing Capability
4c Modified Existing Capability
4d New Capability
5 Certification Test
6,7 Implement / Site Security Test
8 Accreditation (Authorizing Official)
9 Operate & Monitor
10 Appellate process
LOC 1
LOC 2
LOC 3
STOP
STOP

Figure 7. Cross Domain Implementation Process (CDIP)

The outline of Figure 7 is informal in the sense that it pictorially describes a process, yet it has none of the elements necessary to be considered a formal specification. We consider a specification to be formal when it is written with mathematically based techniques as the foundation for the specification language. Gabbar says, "A specification language is based on a set of formulae, written in a formal language, to describe the underlying system" (Gabbar 2006). In the case of our modeling approach it describes the underlying human-based process.

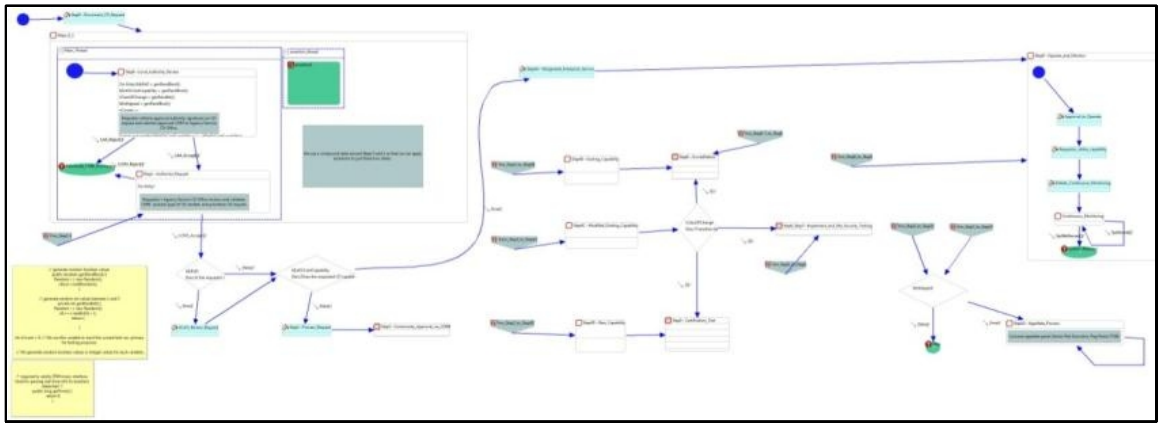


Figure 8. StateRover Model of CDIP

Figure 8 shows a large-scale view of a formal model of the CDIP. We developed this model using our statechart-based formal modeling approach in conjunction with the StateRover modeling tool. The CDIP formal model provides us with an unambiguous view of the process under examination. The process of building the UML statechart-based model serves as the "very important" second look described by Berry and Wing (Berry and Wing 1985).

2. Terminology

We use the terms thread, transition, decision-point, process requirements, timing, complexity, layering, and scenario in a particular way and provide definitions below to ensure readers develop a common understanding of our lexicon for development of a statechart-based process model.

a. Threads

In our approach to formal process modeling, we use statechart threads (the blue dashed-line boxes of Figure 9) to represent orthogonality within states. Conceptually, orthogonality resembles concurrency though it is different in that the activities captured in different threads are, for the most part, independent of, or orthogonal to each other; whereas concurrent activities occur in relation and typically at the same time as one another and due to the interrelationship can involve interference, synchronization, locking, and recovery. Throughout this document, we use the term thread to refer to statechart threads vice OS-level or programming language related threads. The latter threads are typically used as the computer programming implementation of concurrency while the former provides a means for notating the existence of orthogonality within a process.

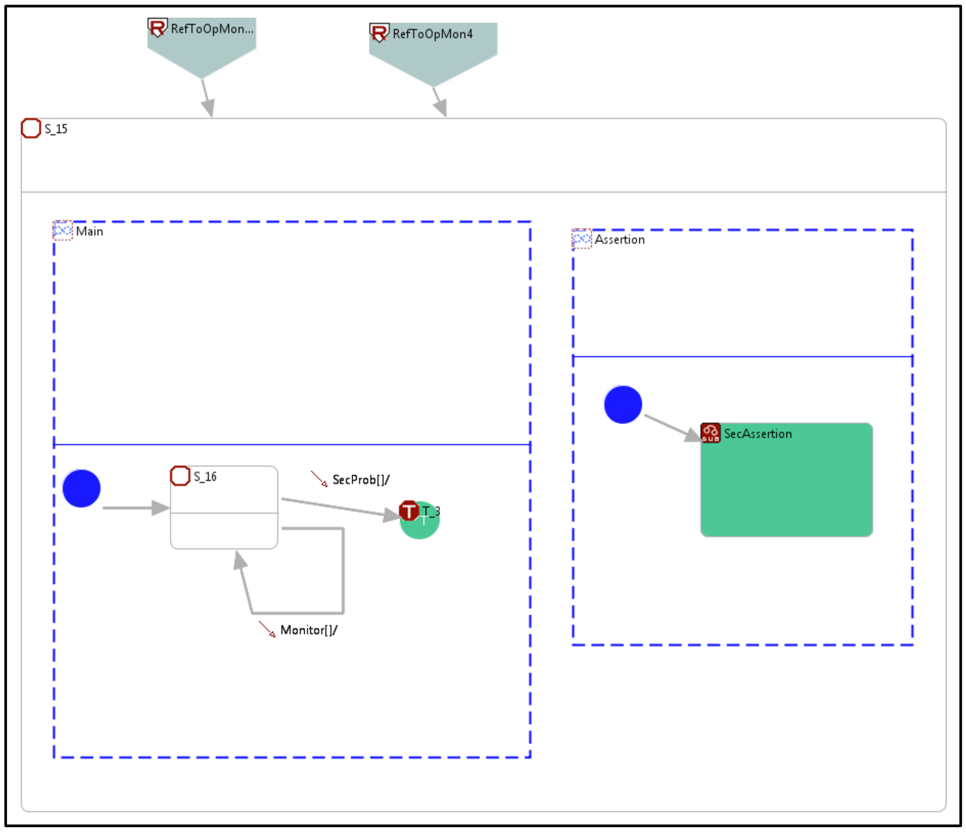


Figure 9. Example of Threads in "Op_Monitor" Sub-Process

b. Transitions

A process has as its atomic operation what we call "steps." There can be sub-processes as well. Sub-processes represent sets of steps grouped together on the basis of cohesion and coherence. The Oxford English Dictionary defines these terms as: cohesion, "the action or condition of cohering; cleaving or sticking together" and coherence, "consistency in reasoning, or relating, so that one part of the discourse does not destroy or contradict the rest" (Oxford English Dictionary 2012a). Within a process there can be transitions between steps, as well as between sub-processes. In our modeling approach, transitions between sub-processes are modeled using an artifact called "off-page references." We see a transition from processing a CDS request to implementing the CDS in Figure 10. *CD_Officer_Validator* is a thread contained within the *Process* sub-process of the CDS model. In this case, we observe a transition from the sub-process named *Process* to the sub-process named *Arbitrate* via the off-page reference *RefToArbitrate1* (outlined in red).

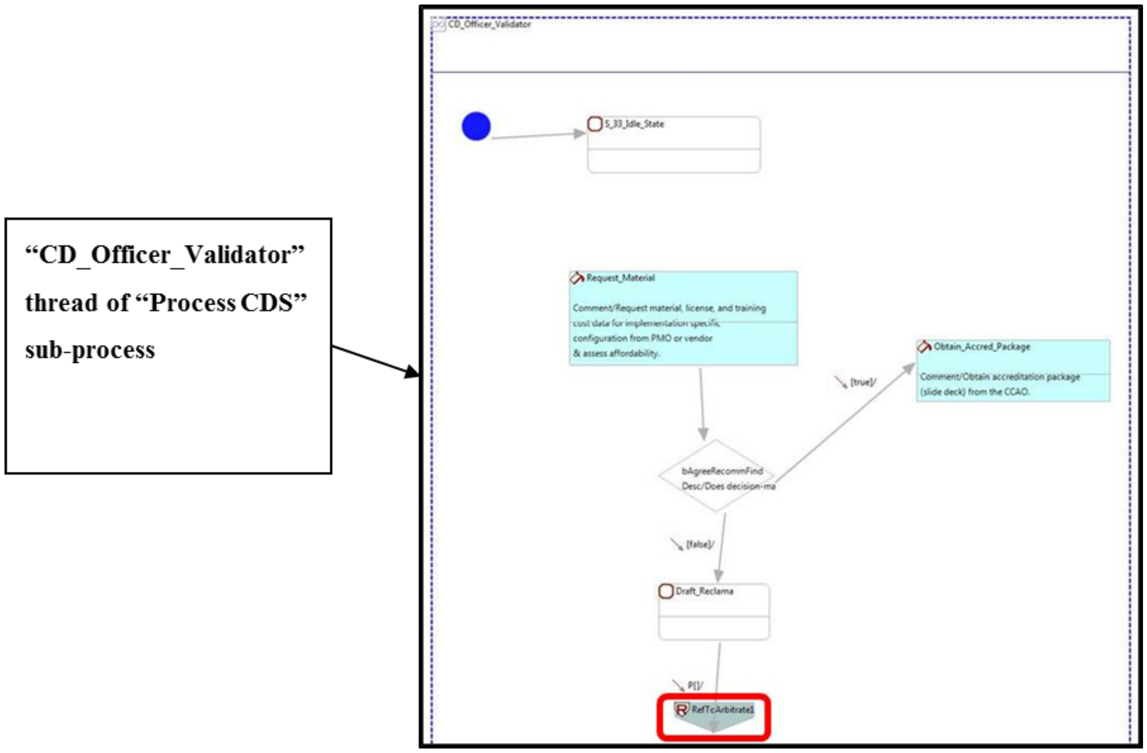

"CD_Officer_Validator" thread of "Process CDS" sub-process
CD_Officer_Validator
S_33_Idle_State
Request_Material
Comment/Request material, license, and training cost data for implementation specific configuration from PMO or vendor & assess affordability.
Obtain_Accred_Package
Comment/Obtain accreditation package (slide deck) from the CCAO.
[true]/
bAgreeRecommFind Desc/Does decision-ma
[false]/
Draft_Reclama
P[]/
RefToArbitrate1

Figure 10. Transition Using an "Off-Page Reference" Artifact

c. *Decision Points*

We are interested in modeling processes that have a strong flavor of the human playing a significant role in the enactment of the process. The process itself can have varying levels of automation, and the level of automation may vary by modality and circumstance. For instance, on a ship, the process of target acquisition and firing may be highly automated under nominal operating conditions, but weapon systems (e.g., 5"/54 series guns) may be operated manually in highly degraded operations. In the context of human-based processes, we refer to decision points as those places where a decision must be made (e.g., yes or no, approve or disapprove) and that decision's outcome falls within an expected range of values. In other words, during process analysis, we seek out the decision points and articulate them in the model as conditional- or value-based transitions between components of the model.

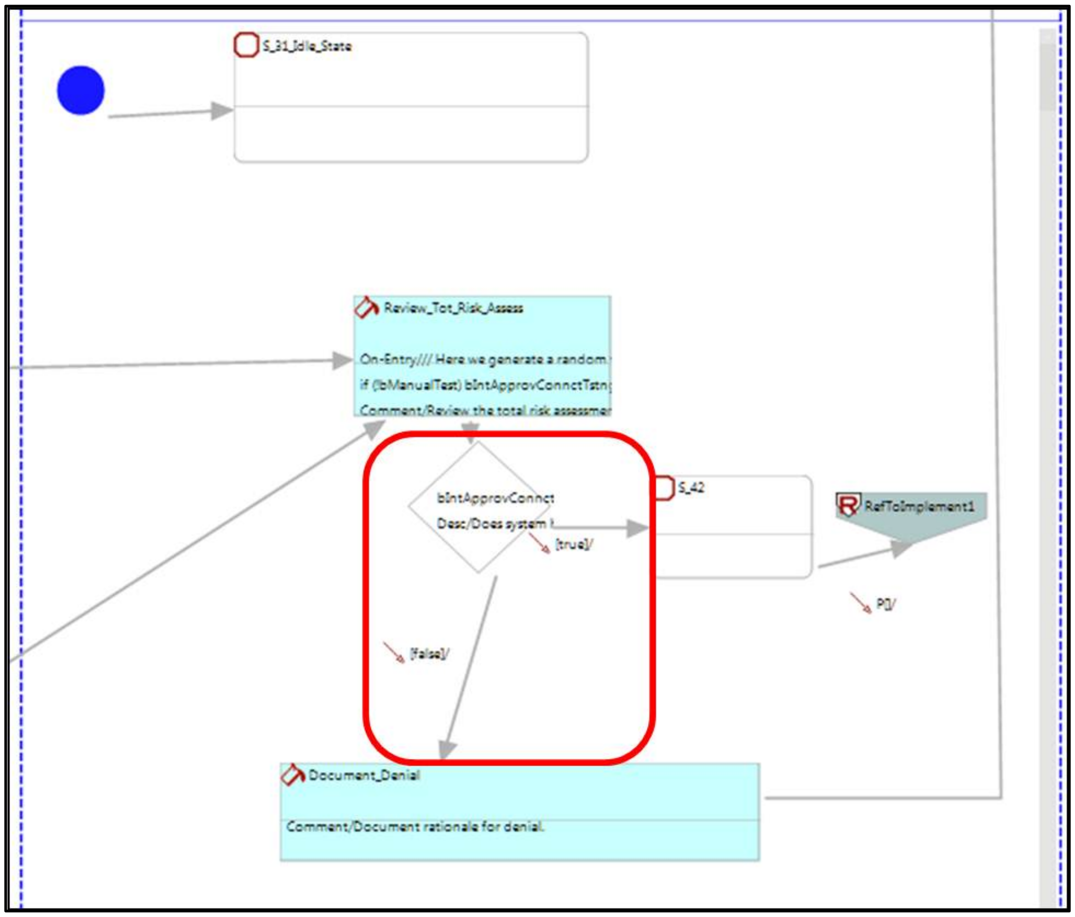

S_31_Idle_State
Review_Tot_Risk_Assess
On-Entry///.Here we generate a random
if (bManualTest) bIntApprovConnctTstn
Comment/Review the total risk assessmer
bIntApprovConnct
Desc/Does system
[true]/
S_42
RefToImplement1
P[]/
[false]/
Document_Denial
Comment/Document rationale for denial.

Figure 11. Decision Point Example

The diamond in Figure 11, labeled *bIntApprovConn* (outlined in red), demonstrates the StateRover artifact view of a decision point in our process model. We found a significant number of decision points when analyzing and modeling our demonstrative exemplar processes. Therefore, we recommend using a table to manage the associated variables and ensure all decision points are implemented in the model as demonstrated in Table 3. In column 1 we use a plain language description of the decision, in column 2 we list the variable name that will be used in StateRover's visual switch construct, in column 3 we list the possible outcomes of the decision point (i.e., the possible values of the variable), and in column 4 we track whether the decision point has been incorporated into the process model.

Table 3. Example Decision Point Tracking

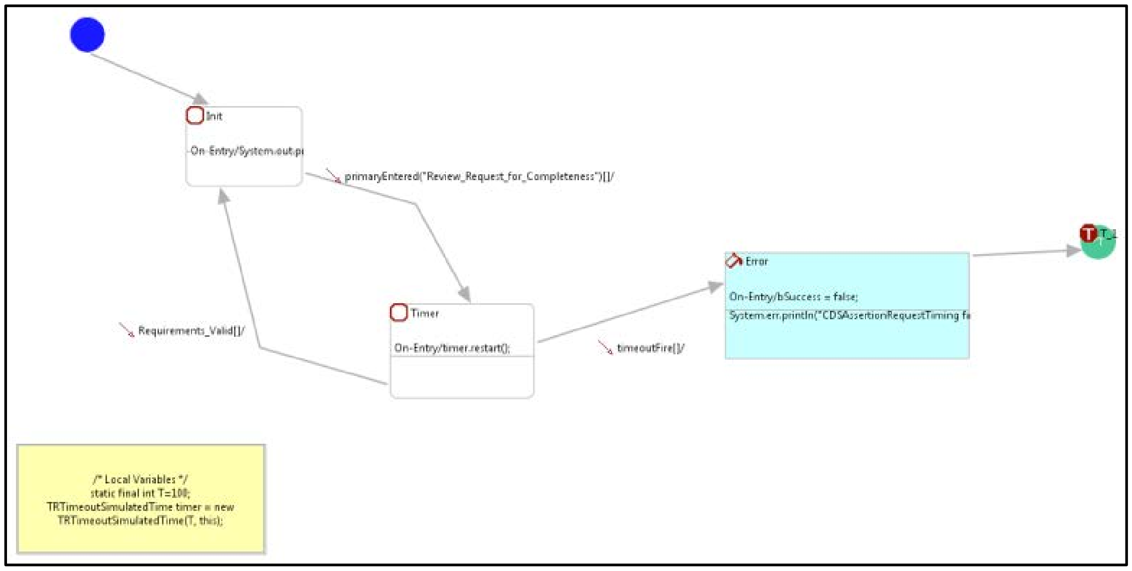
Description	**Variable Name**	**Possible Values**	**In Model**
Decide whether CDS requestor is a DoD component.	bIsDod	True, False	☒
Determine whether capability exists as an enterprise or centralized capability.	bEntOrCentCapaiblity	True, False	☐

d. Process Requirements

Process requirements are those properties, attributes, or timing constraints that must be upheld as the process executes. For example, if a process requires that some event occur at a set time, our process modeling approach provides runtime process monitoring to ensure that the event occurs within a specified temporal constraint. A (formal) specification is a representation of a requirement that uses notation a computer can understand and read in a finite amount of time using finite resources (Drusinsky 2006). Our modeling approach uses a UML statechart-based language to build specifications. We use this formal specification language to create embedded statechart assertions, which serve to enforce process requirements at runtime.

e. *Timing*

Timing refers to quantifiable time constraints or restraints (e.g., the time at which something must occur or the time within with something must occur). In human-based processes, we expect to find many temporal restraints such as deadlines for submitting paperwork. Therefore, it is important for a process modeling approach intended to formalize human-based processes that we are able to capture and effectively address timing related information. In order to build timing into our models, we leverage temporal constructs inherent in our chosen modeling software. For example, the statechart assertion of Figure 12 uses the *TRTimeoutSimulatedTime* construct of StateRover to apply a temporal constraint to the modeled process. We do this by embedding the statechart assertion of Figure 12 into the CDS process model. Embedded statechart assertions are the primary vehicle for building and applying temporal constraints and restraints to a process modeled with our approach.



```
On-Entry/System.out.pr
On-Entry/timer.restart();
On-Entry/bSuccess = false;
System.err.println("CDSAssertionRequestTiming fa
```

```
/* Local Variables */
static final int T=100;
TRTimeoutSimulatedTime timer = new
TRTimeoutSimulatedTime(T, this);
```

Figure 12. Applying Temporal Constraints and Restraints

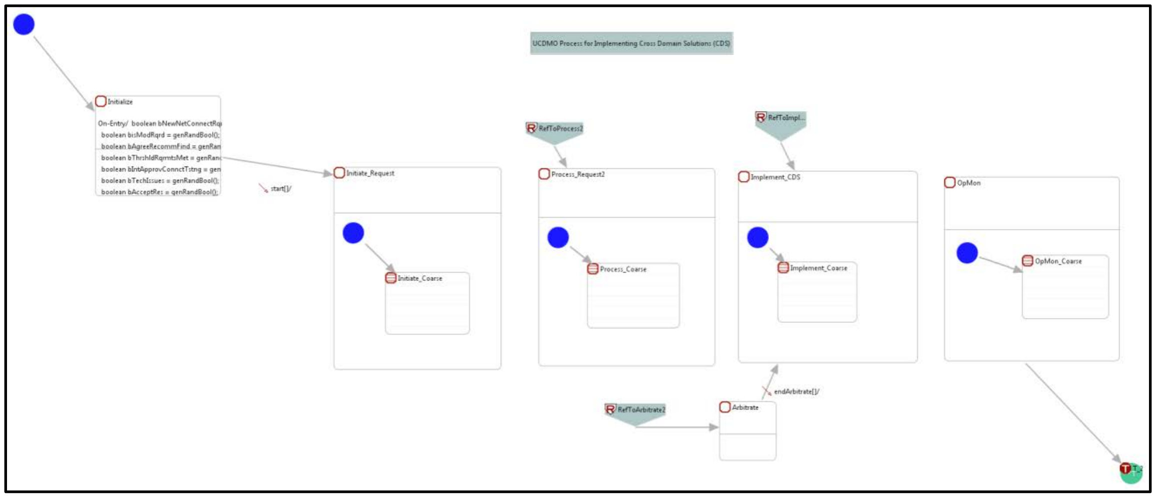
f. *Complexity*

Several researchers have examined the notion of complexity as it relates to processes and have presented metrics to provide information about the understandability and maintainability of business process models (Volker Gruhn and Laue 2006; Cardoso

2007; Cardoso et al. 2006). Cardoso presented the control-flow complexity (CFC) metric for determining the complexity of business processes. This metric is expressed as a summation of joins and splits (AND, OR, or XOR) in a process. In general, the more splits and joins a process has, the more complex it is (i.e., it is more difficult to develop and maintain complex process models). However, as Cardoso points out, the CFC metric is somewhat simplistic and does not account for the increased complexity introduced by nested structures (Cardoso et al. 2006). We adopt Cardoso's definition of process complexity and note that when applied to the type of partially automated, human-based, C&A processes that we are interested in, as the number splits or joins and nested layers increases, so does the complexity of the process.

g. *Layering*

Layering refers to the ability to individually articulate nested levels within a process model. This is a particularly useful technique when analyzing and building models of highly complex processes. For instance; in our exemplar process, the CDS, we wish to capture various views of the model. This approach allows us to view the model at varying levels of complexity, depending on the desired outcome of the viewing. For instance, the top-level view of Figure 13 provides an overview of the process model with each of the major phases depicted as a single state.



```
On-Entry/  boolean bNewNetConnectRq
 boolean bisModRqrd = genRandBool();
 boolean bAgreeRecommFind = genRan
 boolean bThrshldRqmtsMet = genRan
 boolean bIntApprovConnctTstng = gen
 boolean bTechIssues = genRandBool();
 boolean bAcceptRes = genRandBool();
```

Figure 13. Top-level Statechart Model of CDS Workflow Process

This view provides a large-scale aspect on the process vice details of the inner workings of each process. In the second level view, we can examine each sub-process individually as shown in Figure 14 and Figure 15. The initiate sub-process is articulated in Figure 14 while the Op/Monitor is articulated in Figure 15.

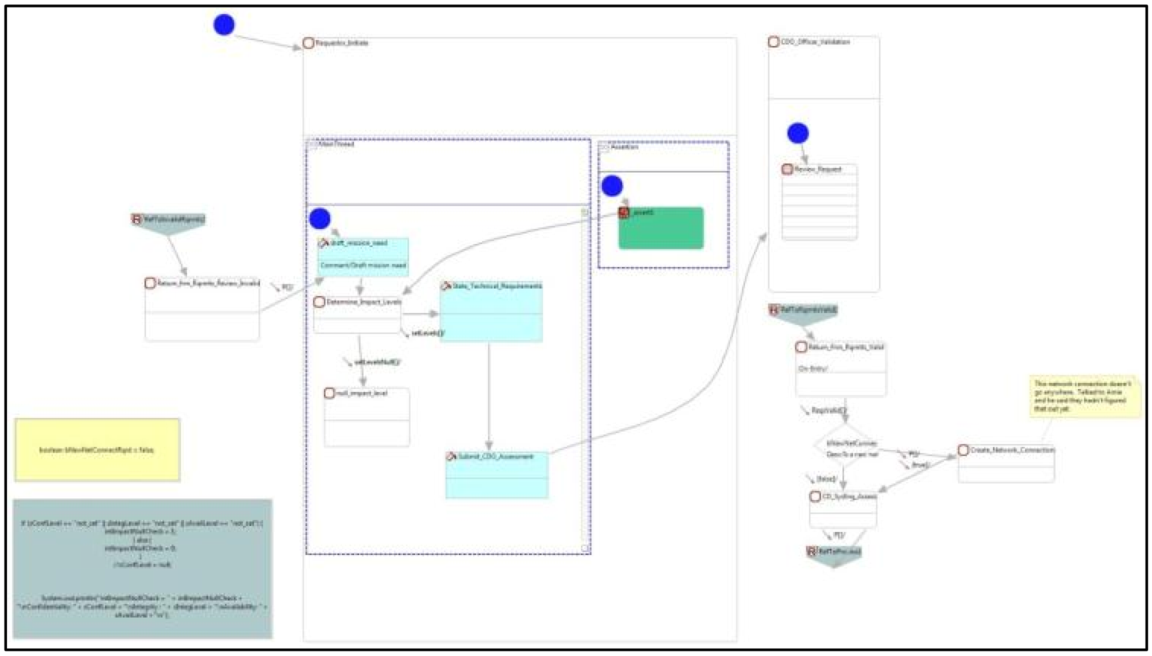


Figure 14. Sub-process Titled “Initiate_CDSR”

These sub-processes sit at the second level of the hierarchy for the purposes of our analysis and each represents a fuller view of its respective sub-process, each of which is represented by a single state at the top-level of the process model. Part of dealing with complexity is the ability to work in the abstract, and then incrementally decompose the process into successively finer levels of detail (i.e., processes, sub-processes, and so on).

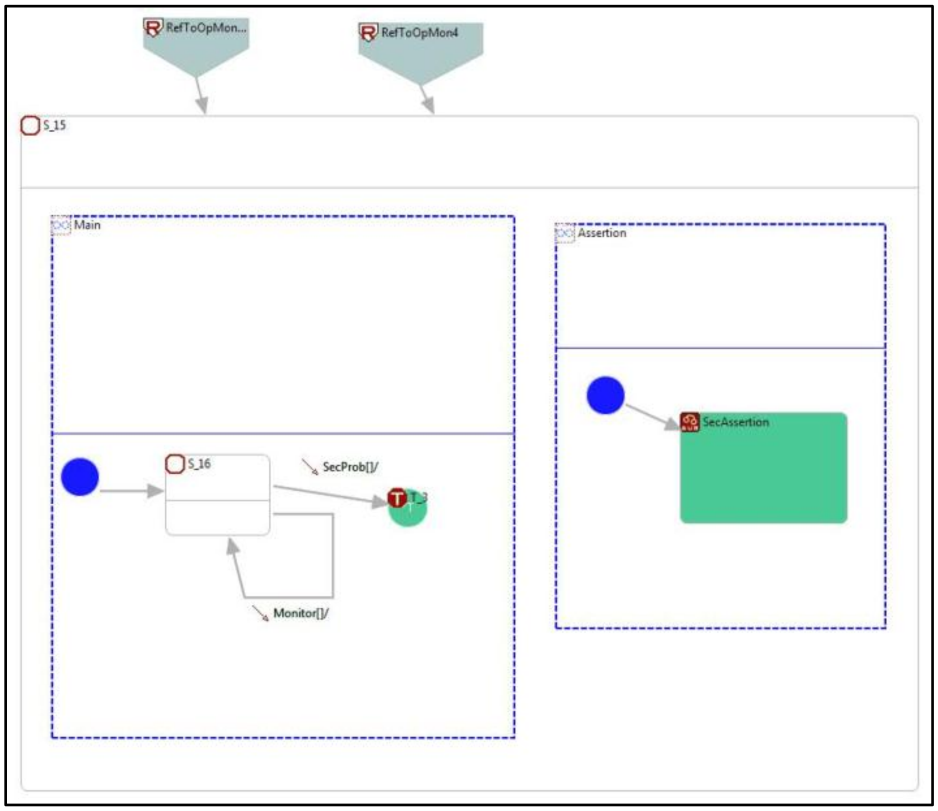


Figure 15. Sub-process Titled “Op/Monitor”

The diagrams shown in Figure 16 demonstrate the successive decomposition of four levels of hierarchy. When viewed together, we see that the ability to deal with complexity in this way makes it possible to formally model hierarchical processes.

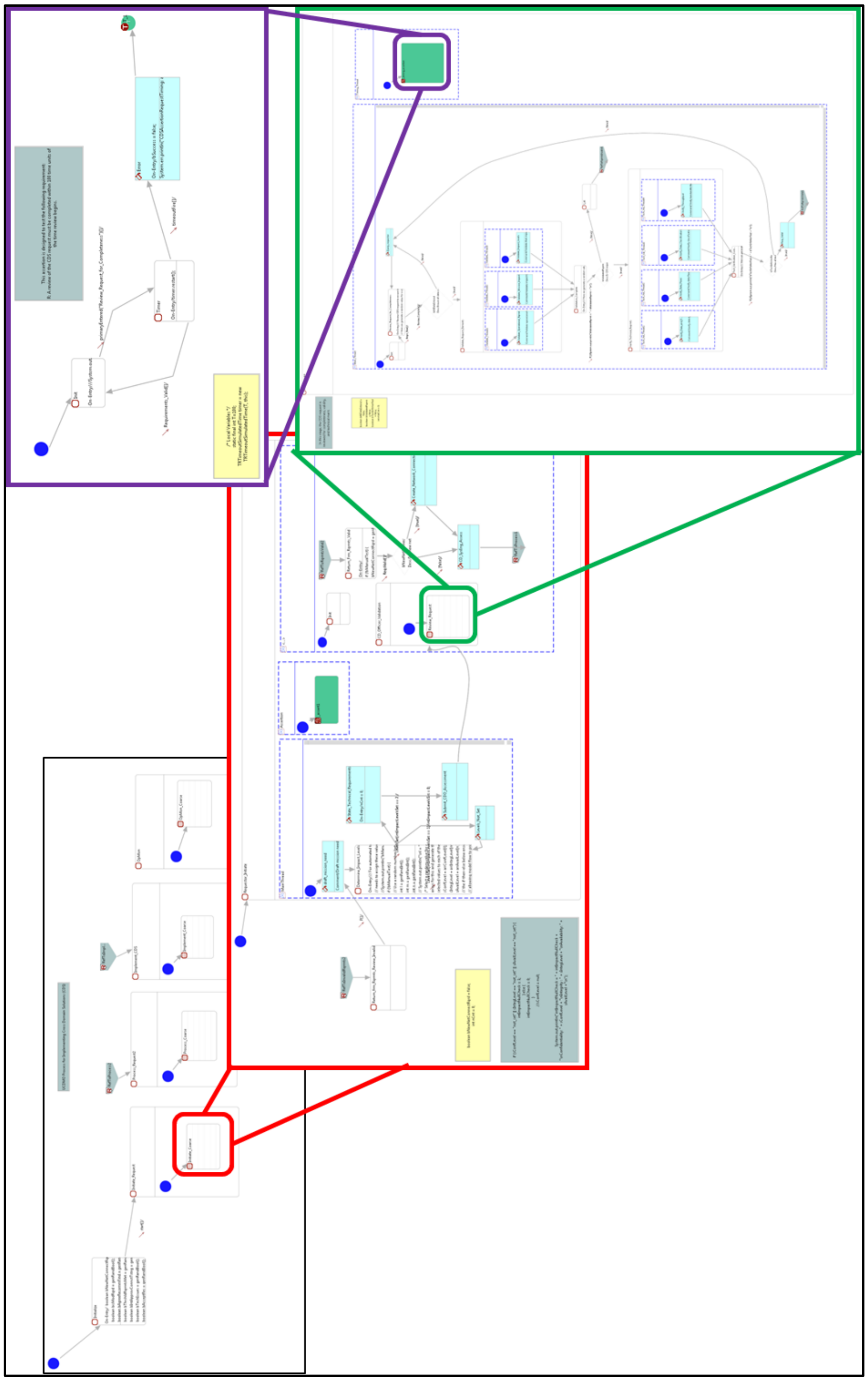


Figure 16. Decomposing a Complex Hierarchical Process Model

h. Scenario

In the context of our modeling approach, the term scenario refers to a combined collection of desired properties, timings, human decisions, and/or conditions that we wish to apply vis-à-vis a process model in order to exercise or stimulate various aspects or behaviors of the model. We use this term to describe situations for which we will develop tests during the "V&V Process Model" step of our modeling approach.

C. MODELING APPROACH

In this section we will discuss the details of each step of our process modeling approach shown in Figure 6.

1. Process Selection

Process selection is a non-trivial matter. Informal processes or well-defined processes whose outcome is not safety or security critical may not require the level of formality afforded by our approach. The application of our statechart-based modeling approach requires a time investment to analyze the process and apply formal methods. However, it is a worthwhile investment for the types of processes that we are interested in formally specifying and reasoning about since the modeled process will be easier to understand and communicate about and the resulting process modeling will be easy to develop, debug, and maintain.

2. Process Analysis

Prior to constructing the process model in StateRover, it is helpful to analyze the process. During this phase of our modeling approach, we are identifying components of the process that lend themselves to articulation as artifacts in a statechart-based formal model. This is also where we identify specific requirements or specifications which we wish the model to adhere to. This eases the process of building and verifying the model. It also facilitates the development of a more robust, granular, and higher fidelity model. Process analysis facilitates a more full-bodied statechart-based process model via the thorough *a priori* inspection of the process during model development. Similarly, process analysis provides the process engineer an opportunity to identify many if not all of the process components and timing considerations with a focus on the behavioral properties of the model vice run of the mill functional properties and requirements. Hence, we are able to develop a more granular and higher fidelity model.

Process formalization requires thorough analysis of the chosen process as a key component of developing the formal model in StateRover. Through analysis, we develop a better understanding of the process under examination and begin to formulate a plan for contextualizing individual components vis-à-vis our modeling approach with its associated views and terminology. We must understand threads, transitions, decision points, process requirements, timing, complexity, layering, and important steps in the process.

The process engineer uses a combination of available sources such as informal drawings, interviews with stakeholders, mission statements, modeling diagrams (e.g., UML activity diagrams, YAWL workflow charts), or basic flowcharts. The focus of this phase of our modeling approach is to develop as complete an understanding of the process as possible. One of the challenges of process analysis is that the stakeholders and/or process owners' understanding and documentation of the process could range from tribal knowledge held by one or a few individuals to more formalized documentation such as written flowcharts or models based on notations like BPMN or UML.

During this phase of the modeling approach the process engineer also gathers requirements from process stakeholders. These will provide the source material for developing embedded statechart assertions, a key element to enable runtime execution monitoring of the process model.

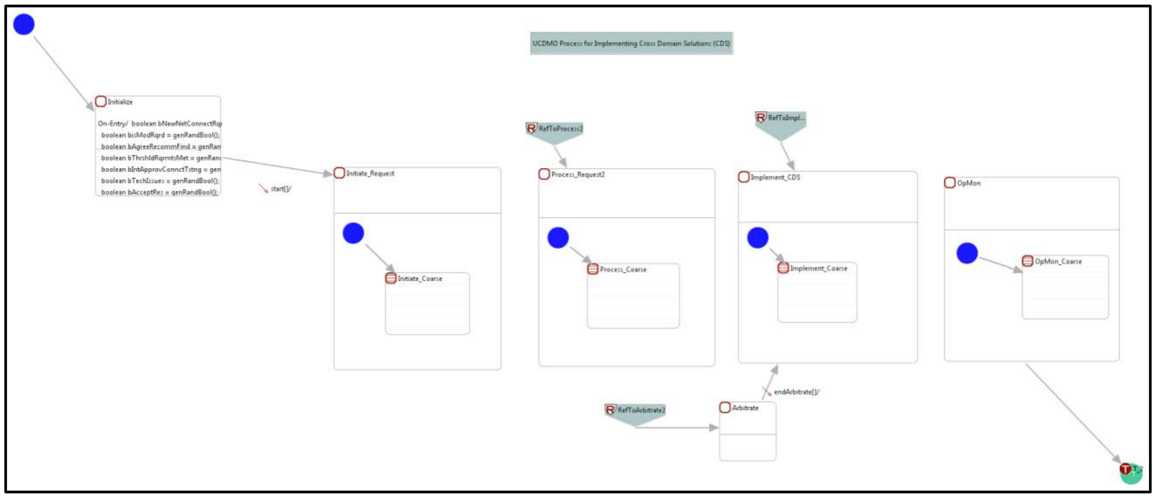
We show in Chapter IV of this document how the analysis of a process leads to a fully realized statechart-based formal model of the process.

3. Construct Process Model

It is during this step of the modeling approach that the process engineer builds the statechart-based process model. Leveraging the products of the "Process Analysis" step he visually articulates the process using the formal language and tools chosen for their adherence to the desirable attributes listed in Section III.A.1.

We previously demonstrated the novel use of UML statecharts as a medium and the StateRover modeling tool as a mechanism for formally modeling the CDIP, a partially automated, human-based, C&A process (Schumann 2009). In this document, we demonstrate the use of UML statecharts as a fundamental component of the process modeling approach shown in Figure 6.

Since our chosen modeling tool generates an executable model in Java we are able to add Java code to just about any component of the model such as states, transitions, or flowchart boxes. In our discussion of case studies, we will show how this functionality helps us ensure that embedded assertions are enforced at runtime.



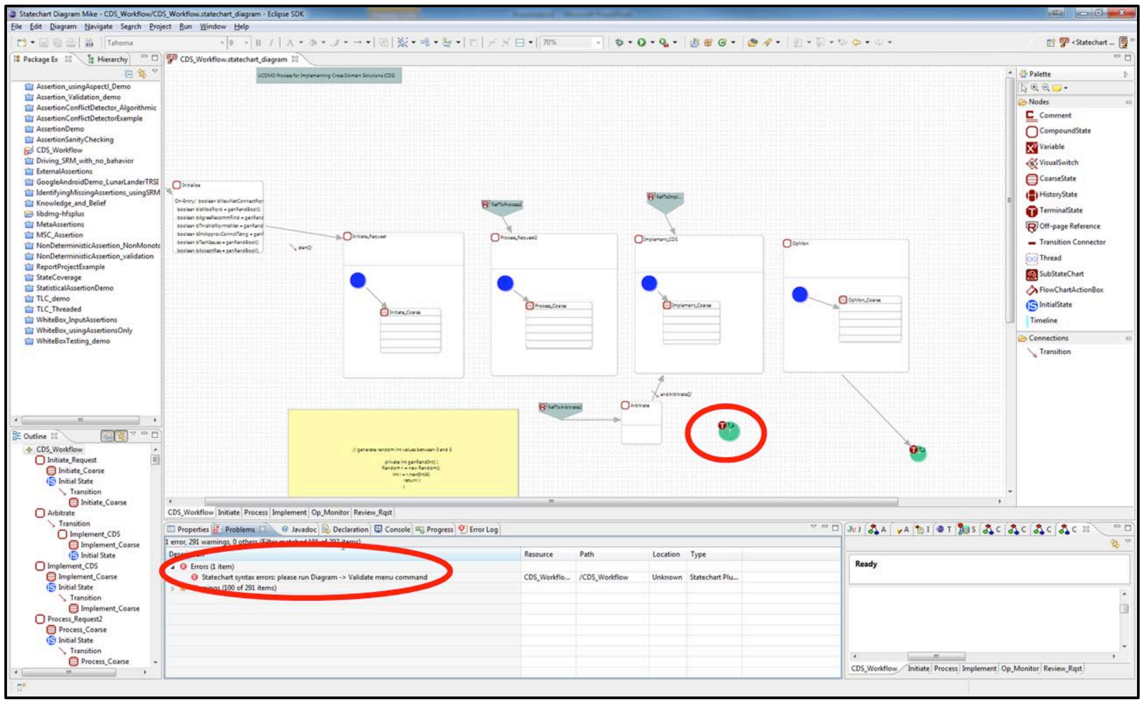
```
On-Entry/  boolean bNewNetConnectRq
 boolean bisModRqrd = genRandBool();
 boolean bAgreeRecommFind = genRan
 boolean bThrshldRqrmtsMet = genRan
 boolean bIntApprovConnctTstng = gen
 boolean bTechIssues = genRandBool();
 boolean bAcceptRes = genRandBool();
```

Figure 17. Top-level Statechart Model of CDS Workflow Process

UML statecharts provide a visually palatable vehicle for the articulation of, formalization of, and communication about a process model such as the one shown in Figure 17, a process model for our demonstrative exemplar, the CDS Workflow process. In addition, we are able to take full advantage of automated statechart-handling capabilities built into StateRover such as hierarchy, concurrency, non-determinism, syntactic validation, workflow modeling, automated testing, and runtime monitoring.

a. Iterative Validation

During the design process, the process engineer is able to use the immediate feedback from StateRover's underlying rule checking mechanisms to identify possible errors within the process model. The process engineer can use the errors identified via this mechanism to diagnose, troubleshoot, and correct inconsistencies in the process model. This systematic approach helps ensure that the model is founded on and adheres to the underlying UML statechart formalisms.



```
1 error, 291 warnings, 0 others (Filter matched 101 of 292 items)
Description                                                      Resource        Path            Location  Type
  Errors (1 item)
    Statechart syntax errors: please run Diagram -> Validate menu command   CDS_Workflo...  /CDS_Workflow  Unknown   Statechart Plu...
  Warnings (100 of 291 items)
```

Figure 18. StateRover Automated Validation

Figure 18 demonstrates the embedded error identification within the StateRover plug-in. For this example, we deliberately placed a unitary *terminal* state, circled in red, in the CDS Workflow process model's top level. Since this terminal state does not have a corresponding *start* state, it constitutes an error and is identified as such via StateRover's embedded validation mechanism. We refer to this as iterative validation and use it throughout construction of the process model and the statechart assertions to ensure they adhere to the underlying UML statechart semantics.

4. Construct Statechart Assertions

In this step, the process engineer transforms the requirements developed during "Process Analysis" into statechart assertions. The UML statechart articulation of each requirement is in the same statechart-based language as the rest of the model. This provides a precise way of stating requirements that directly takes advantage of the formalisms used to develop the process model. During the execution phase of our approach, embedded statechart assertions are employed as enforceable runtime specifications.

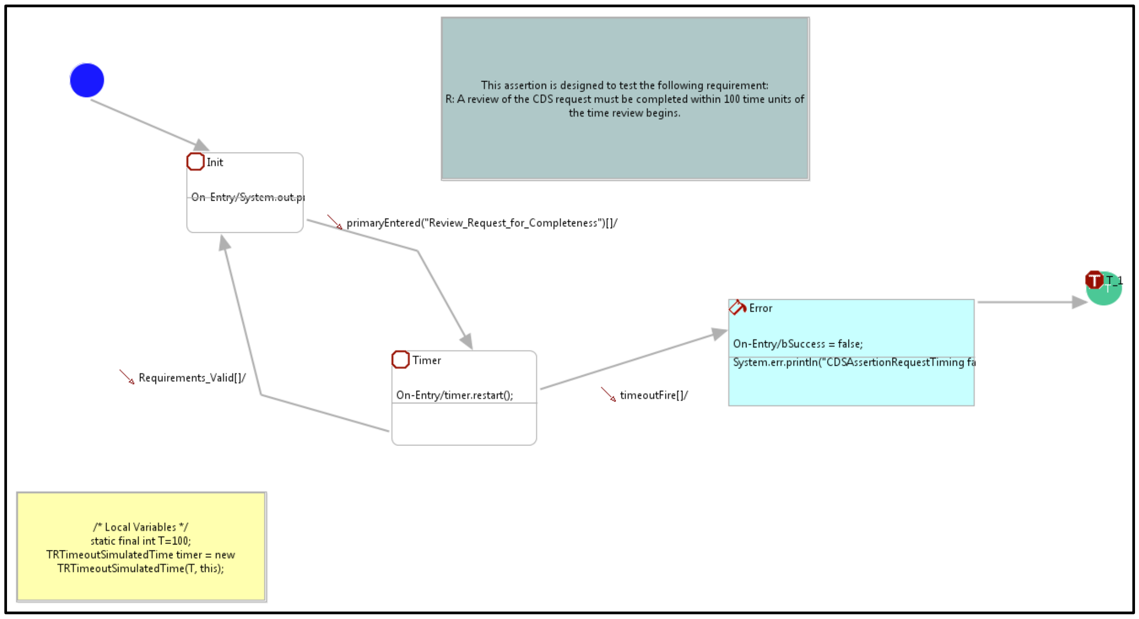
Embedded statechart assertions are a key addition of this research to the process modeling world. They facilitate runtime execution monitoring as well as enforcement of desirable properties or requirements placed on the process (i.e., submission timeline for a request necessary for process progression). Within this section, we cover the conceptual foundations of statechart assertions. We explore their benefits as applied to process engineering and modeling. We investigate their employment to achieve runtime monitoring of a human-based process in execution.

a. Statechart Assertions

A reason for using a formal methods based modeling approach is to demonstrate mathematically that the model adheres to a set of stated requirements. A number of formal notations exist for the specification of formal models. Some examples include the Z Notation, Vienna Development Method (VDM), and the B Method (Monin and Hinchey 2003). Each formal notation can be distinguished by its particular application of set-theoretic mathematical concepts, the underlying logic, or how they assist in the development of computer programs, which is the typical use for such notations. For statechart-based modeling, Drusinsky provided an analogous capability by extending UML statechart diagrams to include statechart assertions, which provide a formal artifact for the specification of requirements (i.e., a formalized specification language).

Statechart assertions have two fundamental differences from the statecharts used throughout the rest of our modeling process. 1) They have a built-in mechanism for indicating Boolean success or failure (true/false), which makes them suitable for formal specification and 2) they can be nondeterministic if desired. Figure 19 Figure 19 shows a statechart assertion.

Drusinsky points out that it is important to exercise meticulous care in the development of statechart assertions as bad assertions reflect poorly conceived requirements and are unlikely to help ensure the system behaves as desired (Drusinsky, Shing, and Demir 2007). Additional papers by Drusinsky, et al. provide more examples of the development and application of embedded statechart assertions (Drusinsky 2008; Drusinsky, Shing, and Demir 2006).


This assertion is designed to test the following requirement:
R: A review of the CDS request must be completed within 100 time units of the time review begins.

```
/* Local Variables */
static final int T=100;
TRTimeoutSimulatedTime timer = new
TRTimeoutSimulatedTime(T, this);
```

Init
On-Entry/System.out.pı
primaryEntered("Review_Request_for_Completeness")[]/
Timer
On-Entry/timer.restart();
Requirements_Valid[]/
timeoutFire[]/
Error
On-Entry/bSuccess = false;
System.err.println("CDSAssertionRequestTiming fa
T_1

Figure 19. Example of an Statechart Assertion

Sindre and Opdahl postulate that a visually appealing approach may actually be more successful than a textual approach when capturing requirements. This is because simple and intuitive diagrams provide a better overview of the functionality of a system and make it easier to see each stakeholder's interest in the system which makes it easier to communicate about the captured requirements (Sindre and Opdahl 2000). The combination of UML statecharts and embedded statechart assertions provides us with a visually appealing formal process modeling approach wherein the model and the assertions that enforce properties of the model are written in the same language, in this case UML statecharts. This addresses one of the challenges seen in previous formal process modeling research; the integration of a separate formal specification language in order to add formalisms to the process modeling approach (Emmerich and Gruhn 1991). Additionally, process models will be easier to develop, debug, and maintain due to the ability for users, stakeholders, and process engineers to easily communicate about the modeled process.

b. Validating Statechart Assertions

According to the Oxford English Dictionary, an oracle is, "an opinion or declaration regarded as authoritative and infallible" (Oxford English Dictionary 2012b). Since both our process model and the statechart assertions that represent requirements on the model are derived from natural language descriptions, it cannot be assumed that one is more of an oracle than the other. However, the properties upon which we base statechart assertions are typically small enough that they don't require more than five to ten validation tests. This suggests that we can use a relatively small number of tests to build a body of evidence for using the assertions as an oracle for testing the behaviors of a process model. A pattern-based methodology like the one described by Drusinsky, Michael, Otani, and Shing can help ensure that we cover often overlooked testing areas when writing validation tests for our assertions (Drusinsky et al. 2008; Drusinsky 2011). They describe scenario test patterns such as *obvious success*, *obvious failure*, *full scenario success, full scenario failure*.

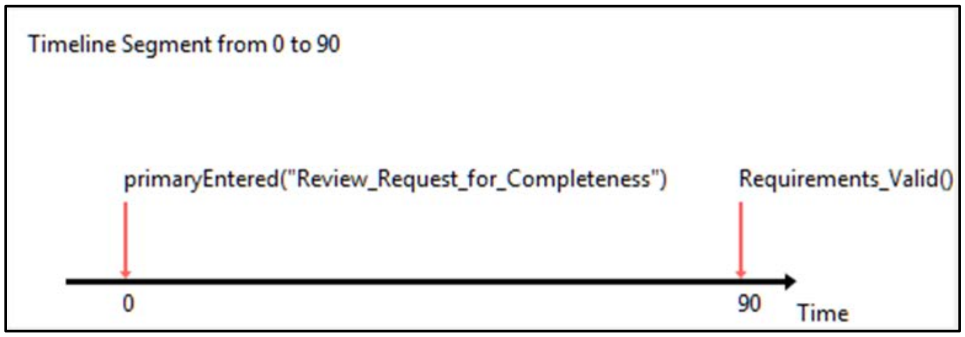

Timeline Segment from 0 to 90
primaryEntered("Review_Request_for_Completeness")
Requirements_Valid()
0
90
Time

Figure 20. Timeline Diagram for "*obvious success*" Assertion Test Scenario

The diagram of Figure 20 shows a test scenario to ensure that the assertion of Figure 19 succeeds when it is supposed to for a simple set of conditions (i.e., obvious success test pattern). Once the assertion has entered state *Timer* it must see a *Requirements_Valid()* event with 100 time units or the assertion fails. The event *Requirements_Valid()* occurs at time 90 and no further events or transitions occur so we expect this assertion succeed during testing.

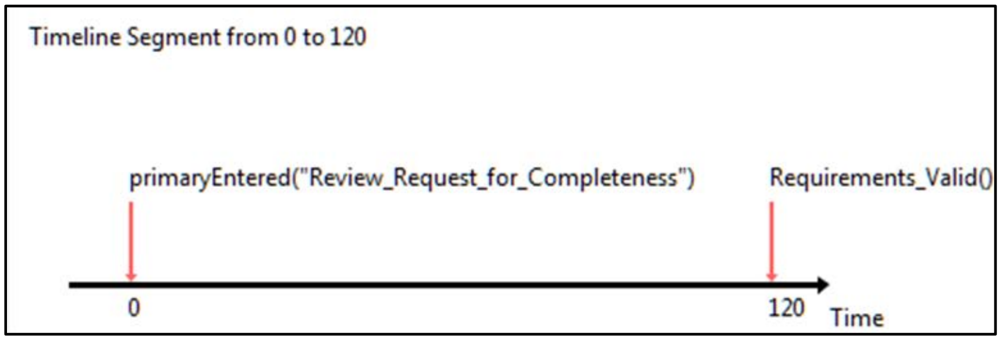

Timeline Segment from 0 to 120
primaryEntered("Review_Request_for_Completeness")
Requirements_Valid()
0
120
Time

Figure 21. Timeline Diagram for "*obvious failure*" Assertion Test Scenario

In contrast Figure 21 shows a test scenario to ensure that the assertion of Figure 19 fails when it should for a simple set of conditions (i.e., obvious failure test pattern). In this case, the timer advances past 100 prior to a *Requirements_Valid()* event so a timeout will fire and cause the assertion to fail.

5. Embed Assertions in Process Model

During this step of the modeling approach, representative artifacts for each statechart assertion are embedded in the process model. Figure 22 demonstrates the use of an embedded assertion in the CDS Workflow formal process model. In the "Initiate CDS Request" phase of the CDS Workflow we use an embedded assertion, *_assert1* (see Figure 22, outlined in red), to ensure that the human decision makers initiating a CDS request have set impact levels for the requested CDS. Impact levels provide a means of categorizing national security systems in terms of the potential impact of unauthorized disclosure of the information residing on the system and must be explicitly stated as part of a CDS request.

We previously examined testing of statechart-based formal process models and showed that embedded assertions could be applied to formal models of human-based processes (Schumann and Michael 2009) as a means of enforcing requirements. When testing the model, failures to adhere to the requirements of the assertion are recorded and reported by the testing module. This ensures that the model behaves as expected under a wide variety of conditions while the executable version of the model is running. This technique allows us to use embedded assertions as an enforcement tool for process requirements because the embedded assertions are located within the model (see Figure 22), which provides unique access to the process model's events, variables, and timing structures as it executes. This positioning is the enabler that allows embedded statechart assertions to act in an enforcement role.

The *_assert1* box of Figure 22 is an example of the method by which an statechart assertion is embedded within a process model. The *_assert1* box acts as a placeholder and insertion point for the statechart assertion of Figure 19 which shows the statechart assertion for ensuring that meets temporal requirements. The natural language version of this requirement is:

"R1: A review of the CDS request must be completed within 100 time units of the time review begins."

If this assertion detects a setLevelsNull system event, bSuccess is set to false whereas a setLevels system event prints a message to the runtime monitor console and the statechart assertion remains in the *Start* state. This provides one example of the type of response mechanism available upon detection of a system event. The bSuccess

Boolean variable allows the process engineer to validate whether the conditions of the assertion have been met or not. Verification occurs through an interlacing of the process model and the JUnit Test framework to apply a variety of automatically and manually generated testing scenarios.

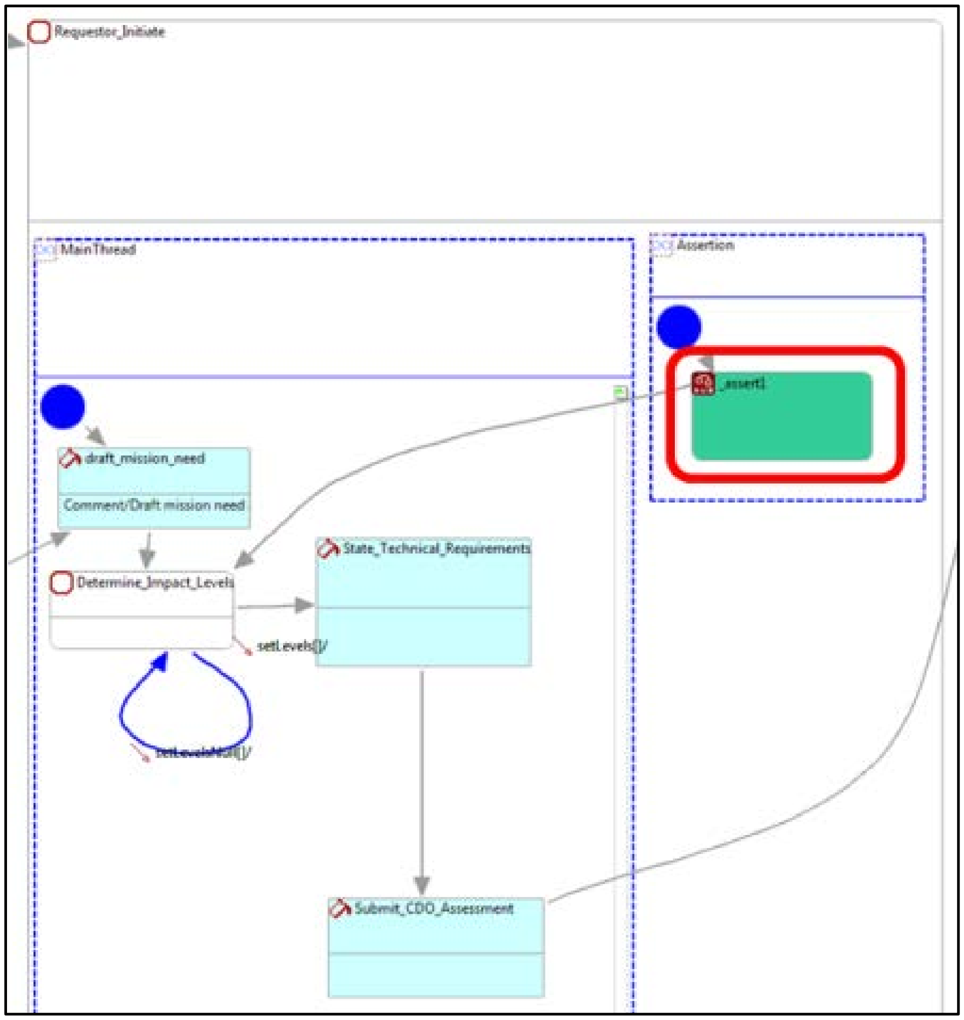


Figure 22. Statechart Assertion Scoped by Substatechart Requestor_Initiate

An added advantage of embedded statechart assertions is that they are naturally scoped by the context of their substatechart (Drusinsky 2006). Therefore, they are only active when their substatechart is entered and they cease to be active when the process transitions out of the containing substatechart. This property of embedded statechart assertions lends itself to hierarchy and scalability in the process modeling approach. This property also allows the process engineer to better deal with process complexity by providing the ability to enforce process requirements in runtime at a variety of levels and with varying scope. This also facilitates easier development, debugging, and maintenance as the process engineer can quickly ascertain the scope of each embedded statechart assertion.

6. V&V Process Model

During this step of the modeling approach, we validate and verify the process model. In order to discuss the notion of validating and verifying a process model we must first define the terms validation and verification. Our research applies the terms as defined by Drusinsky, Michael, and Shing (Drusinsky, Michael, and Shing 2007). Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose. Simply stated, validation asks the question, “Did we build the right product?” Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Simply stated, verification asks the question, “Did we build the product right?” We leverage the components and capabilities of our chosen formal modeling tool to validate and verify statechart-based process models.

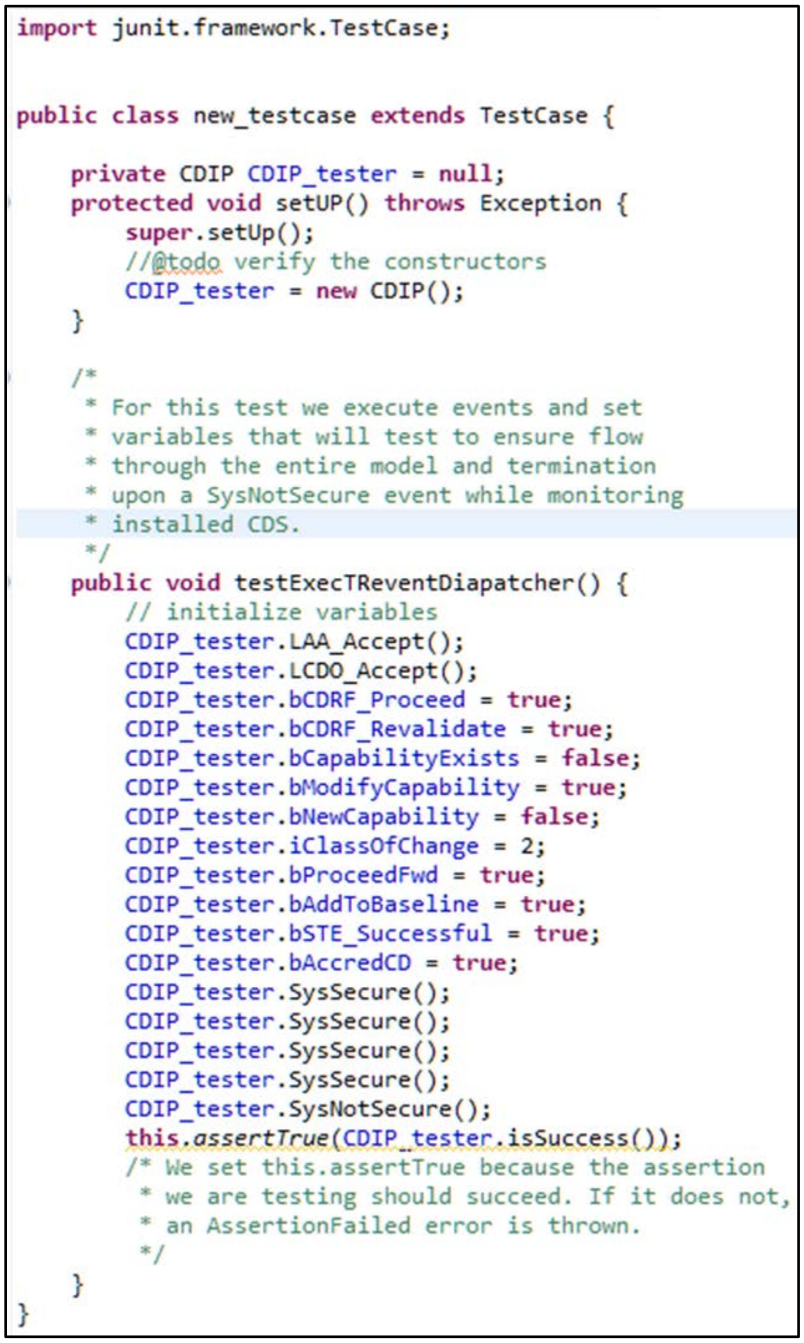
We employ two types of verification in the modeling approach, manual testing and runtime execution monitoring. Each of these uses test cases which are an important construct for verifying that the executable version of a process model operates as intended. Test cases are created in one of two ways. They may be manually generated by the process engineer or automatically generated via the StateRover code generation module and these two types of test cases are used in the “Manual Testing” and “Runtime Execution Monitoring (Automated testing)” steps of our modeling approach, respectively. Both automatically and manually generated test cases represent encoded version of testing scenarios. For the type of partially automated, human-based, C&A processes that we are interested in modeling, test cases equate to sequences of real-world process related events, conditions, timing, and human decisions. They allow us to examine the response or flow of a process model in a simulated test environment.

a. Validation

Our modeling approach uses three types of validation. 1) Automated syntactic validation via algorithms built into our chosen modeling tool which we discussed in Section III.C.3.a. 2) Validation of statechart assertions to ensure they accurately represent stakeholder requirements 3) Validation against stakeholder expectations which requires process engineers to compare the model to requirements derived during the "Process Analysis" phase of our approach and to maintain a review and feedback loop with process stakeholders. The intent of validation is to ensure the process model remains synchronized with stakeholder expectations throughout the design process. In the Figure 6 overview of our modeling approach, this type of validation is represented by dashed lines showing the feedback loop from process model to stakeholders.

b. Verification – Manual Testing

The process engineer uses manually generated test cases for multiple purposes. During the "Construct Process Model" and "Construct Statechart assertions" steps of process model development, he writes manual tests to iteratively ensure components of the model behave as expected. He also uses manual tests to ensure the model, as a whole, accurately reflects the process being modeled and that the process produces expected results for specific scenarios. He also builds tests to examine one or more portions of the process during development, debugging, or maintenance. Figure 23 shows a manually generated test case that represents a testing scenario for the CDIP.



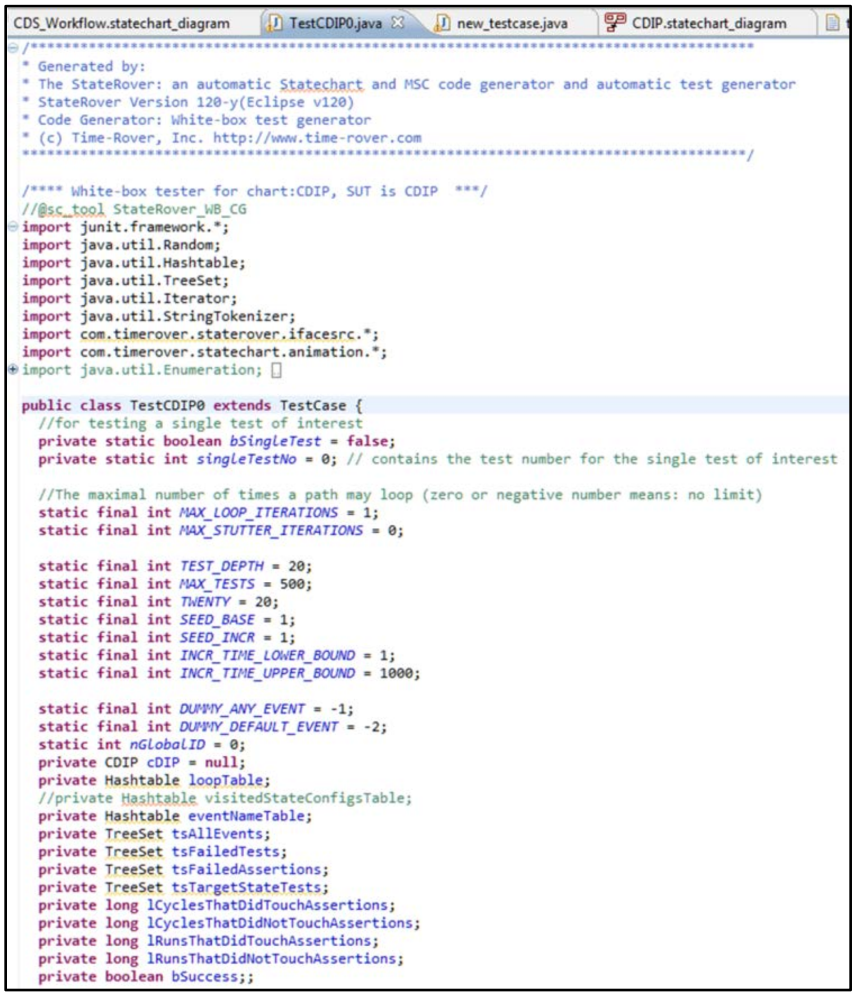
```
import junit.framework.TestCase;


public class new_testcase extends TestCase {

    private CDIP CDIP_tester = null;
    protected void setUP() throws Exception {
        super.setUp();
        //@todo verify the constructors
        CDIP_tester = new CDIP();
    }

    /*
     * For this test we execute events and set
     * variables that will test to ensure flow
     * through the entire model and termination
     * upon a SysNotSecure event while monitoring
     * installed CDS.
     */
    public void testExecTReventDiapatcher() {
        // initialize variables
        CDIP_tester.LAA_Accept();
        CDIP_tester.LCDO_Accept();
        CDIP_tester.bCDRF_Proceed = true;
        CDIP_tester.bCDRF_Revalidate = true;
        CDIP_tester.bCapabilityExists = false;
        CDIP_tester.bModifyCapability = true;
        CDIP_tester.bNewCapability = false;
        CDIP_tester.iClassOfChange = 2;
        CDIP_tester.bProceedFwd = true;
        CDIP_tester.bAddToBaseline = true;
        CDIP_tester.bSTE_Successful = true;
        CDIP_tester.bAccredCD = true;
        CDIP_tester.SysSecure();
        CDIP_tester.SysSecure();
        CDIP_tester.SysSecure();
        CDIP_tester.SysSecure();
        CDIP_tester.SysNotSecure();
        this.assertTrue(CDIP_tester.isSuccess());
        /* We set this.assertTrue because the assertion
         * we are testing should succeed. If it does not,
         * an AssertionFailed error is thrown.
         */
    }
}
```

Figure 23. Manually Generated Test Case Used in CDIP Verification

This test case executes a scenario to test flow through the model to ensure that specific events and variable settings will cause the model to behave in the way we expect. In this case, we wish to see movement through the entire model via a particular path and termination of model execution if a *SystNotSecure()* event is detected during the operation and monitoring phase of a CDS' lifecycle.

c. Verification – Runtime Execution Monitoring

During the manual testing phase, the process engineer works at a micro scale, setting variables and events to an executing process model via handwritten test cases. In contrast, during automated testing he works at the macro scale, adjusting test parameters such as number of tests, test length, number of permissible loops or choosing between stochastic and deterministic testing algorithms while a white-box test generator (WTG) automatically adjusts the micro scale elements at runtime. He is able to use the information from testing to better understand, debug, maintain, and communicate about a process model. The executable version of a process model provides a vehicle for runtime execution monitoring. The medium within which this vehicle operates is the JUnit test framework. By leveraging the JUnit test framework we are able to apply at runtime automatically or manually generated test cases against the executable representation of our model. Automatically generated test cases facilitate exploration of all possible execution paths available to the executable model and exploration of the effect of numerous input sequences on the process model.



```
/*************************************************************************************
 * Generated by:
 * The StateRover: an automatic Statechart and MSC code generator and automatic test generator
 * StateRover Version 120-y(Eclipse v120)
 * Code Generator: White-box test generator
 * (c) Time-Rover, Inc. http://www.time-rover.com
 *************************************************************************************/

/**** White-box tester for chart:CDIP, SUT is CDIP  ***/
//@sc_tool StateRover_WB_CG
import junit.framework.*;
import java.util.Random;
import java.util.Hashtable;
import java.util.TreeSet;
import java.util.Iterator;
import java.util.StringTokenizer;
import com.timerover.staterover.ifacesrc.*;
import com.timerover.statechart.animation.*;
import java.util.Enumeration;

public class TestCDIP0 extends TestCase {
  //for testing a single test of interest
  private static boolean bSingleTest = false;
  private static int singleTestNo = 0; // contains the test number for the single test of interest

  //The maximal number of times a path may loop (zero or negative number means: no limit)
  static final int MAX_LOOP_ITERATIONS = 1;
  static final int MAX_STUTTER_ITERATIONS = 0;

  static final int TEST_DEPTH = 20;
  static final int MAX_TESTS = 500;
  static final int TWENTY = 20;
  static final int SEED_BASE = 1;
  static final int SEED_INCR = 1;
  static final int INCR_TIME_LOWER_BOUND = 1;
  static final int INCR_TIME_UPPER_BOUND = 1000;

  static final int DUMMY_ANY_EVENT = -1;
  static final int DUMMY_DEFAULT_EVENT = -2;
  static int nGlobalID = 0;
  private CDIP cDIP = null;
  private Hashtable loopTable;
  //private Hashtable visitedStateConfigsTable;
  private Hashtable eventNameTable;
  private TreeSet tsAllEvents;
  private TreeSet tsFailedTests;
  private TreeSet tsFailedAssertions;
  private TreeSet tsTargetStateTests;
  private long lCyclesThatDidTouchAssertions;
  private long lCyclesThatDidNotTouchAssertions;
  private long lRunsThatDidTouchAssertions;
  private long lRunsThatDidNotTouchAssertions;
  private boolean bSuccess;;
```

Figure 24. White-box Test Generator Code Snippet

StateRover employs an embedded WTG, which is automatically created by the code generation model. The WTG of the CDIP process model is shown in Figure 24. This provides the flexibility to generalize and scale the WTG to a wide variety of processes. The WTG is specific to each SUT and is built during automated code generation of the executable version of the process model.

Table 4. Runtime Execution Monitoring Data Collection

Description of Attributes	*Test Run #*			
	1	**2**	**3**	**4**
Number of tests per run	5	25	50	100
Failed assertions	☑	☑	☑	☑
All states visited	☐	☐	☐	☑
Time to complete test run	77.5s	398.2s	853.9s	1523.6s

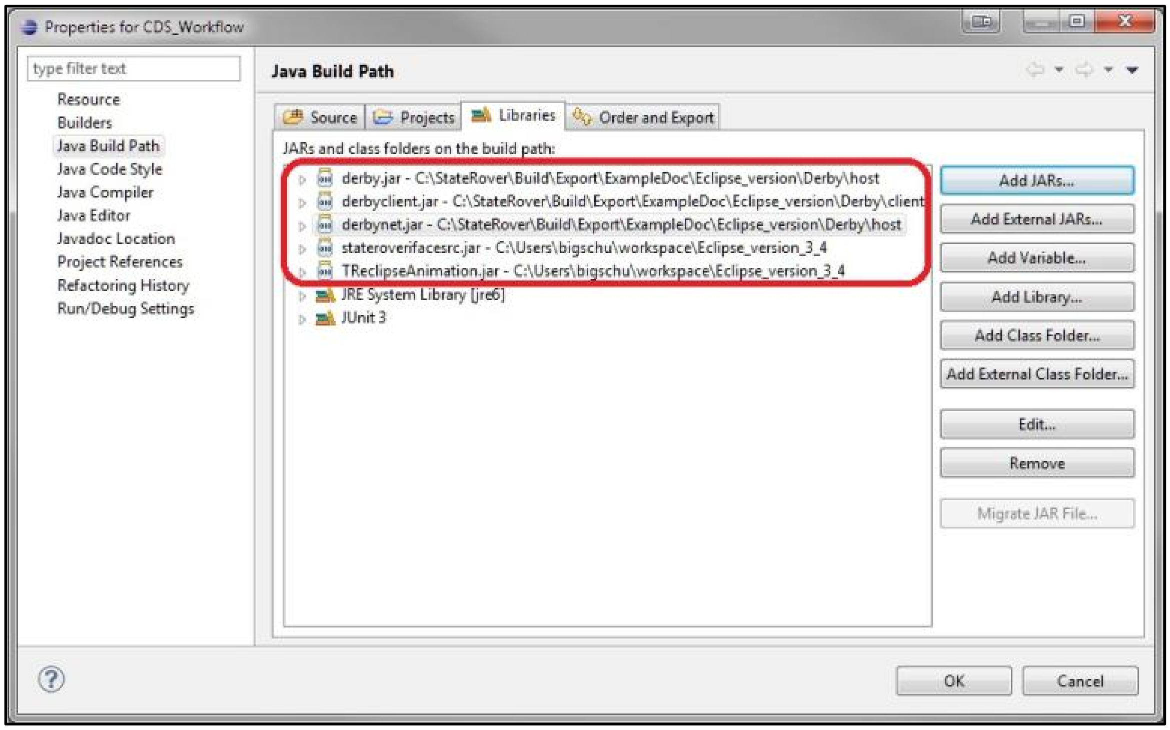
We use the feedback from testing to help assess and compare things such as number of tests per test run, whether assertions failed in a test run, whether or not all states were visited during testing, and time to complete each test run. These factors can be tabulated and compared via a table format as shown in Table 4. A process engineer should collect and compare the data necessary to understand, debug, and maintain the process model. In our case, we show Table 4 as an example of the type of data we found useful while developing our case studies.

D. STATEROVER MODELING TOOL

In this section we provide an overview of how the StateRover modeling tool, in conjunction with the JUnit testing framework, can be used to carry out our modeling procedure and list some of the technical details and considerations when using the tool. As detailed in Section III.A.1, StateRover was chosen because it most closely matched the set of desirable attributes for a tool that would facilitate the development of process models in our chosen formal language.

1. Adding .Jar Files

When setting up the StateRover for process model development, several key .jar files must be added to the Java Build Path found in the project properties as shown in Figure 25, circled in red. Stateroverifacesrc.jar and TReclipseAnimation.jar are included with the StateRover plugin and are required for code generation and animation of process models created with StateRover. The files derby.jar, derbynet.jar, and derbyclient.jar are required to enable StateRover's data collection and reporting capabilities via an embedded or external data collection facility. The process engineer uses the "Add external JARs" command of the Java Build Path window to add these .jar files as a component of the project. This step needs to be taken for each StateRover project.



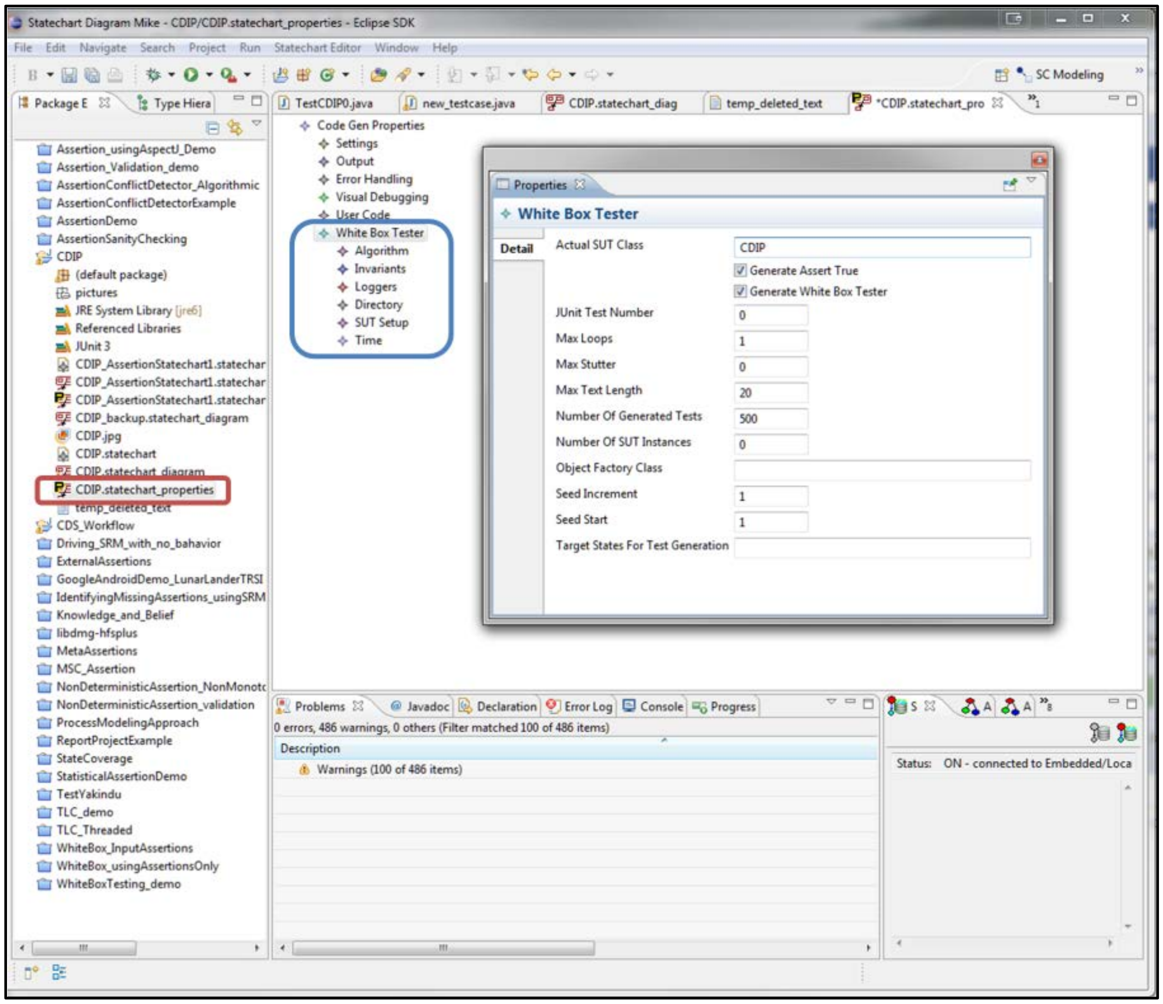
JARs and class folders on the build path:

- derby.jar - C:\StateRover\Build\Export\ExampleDoc\Eclipse_version\Derby\host
- derbyclient.jar - C:\StateRover\Build\Export\ExampleDoc\Eclipse_version\Derby\client
- derbynet.jar - C:\StateRover\Build\Export\ExampleDoc\Eclipse_version\Derby\host
- stateroverifacesrc.jar - C:\Users\bigschu\workspace\Eclipse_version_3_4
- TReclipseAnimation.jar - C:\Users\bigschu\workspace\Eclipse_version_3_4
- JRE System Library [jre6]
- JUnit 3

Figure 25. Adding Necessary .jar Files to Java Build Path

2. Setting up the White-box Test Generator

In order to facilitate white box testing, the process engineer is able to adjust the parameters of the white-box test generator (WTG) embedded in StateRover. These parameters are adjusted via the "statechart.properties" file created automatically for each new statechart diagram. For example, in Figure 26 the “CDIP” statechart properties filename is circled in red with its associated WTG properties circled in blue.



Actual SUT Class: CDIP
[✓] Generate Assert True
[✓] Generate White Box Tester
JUnit Test Number: 0
Max Loops: 1
Max Stutter: 0
Max Text Length: 20
Number Of Generated Tests: 500
Number Of SUT Instances: 0
Object Factory Class:
Seed Increment: 1
Seed Start: 1
Target States For Test Generation:

0 errors, 486 warnings, 0 others (Filter matched 100 of 486 items)
Warnings (100 of 486 items)
Status: ON - connected to Embedded/Loca

Figure 26. Statechart Properties and WTG Parameters

The “White Box Tester” properties view shown in Figure 26 demonstrates some of the adjustable parameters. In many cases, the default properties applied at code generation time are sufficient. For more advanced or complex process modeling, the process engineer has the flexibility to adjust parameters via this mechanism.

3. JUnit Testing Framework

JUnit is well suited to the enable automated verification of partially automated, human-based, C&A processes. Conceptually, the JUnit test framework is a pattern-based framework of programs designed to facilitate the testing of software program components. It allows the programmer to write scenarios to be implemented as automated JUnit tests. One of the advantages of this approach is that once tests are written they are repeatable, long lasting, and available for use in other testing situations or modified versions of the original testing scenarios. This enables developers to iteratively improve both the program under development and the automated tests used to ensure that the program functions as desired/required. Vlissides states that JUnit has three primary goals. One, provide a framework within which developers will actually write tests. Incorporating common developer tools into JUnit does this. Two, allow test writers to create tests that retain their value over time. JUnit does this through Java based test scenarios that, once written, can be understood and used by other process engineers. Three, it has to be possible to leverage existing tests to create new ones (Vlissides, Johnson, and Edgar 2011). Again, JUnit facilitates this through Java-based test scenarios that, once created, can be used as the basis for additional scenarios.

In the context of our statechart-based approach to process modeling, the process engineer uses the JUnit testing framework to test components of the process model or the complete process model. JUnit provides a vehicle for runtime execution monitoring of the StateRover generated executable representation of the process model.

StateRover fully integrates the use of statechart assertions. Drusinsky describes three ways of applying assertions: as a component of the testing process, as part of a simulation, and as a component of runtime execution monitoring (Drusinsky 2006, 229–230). All three of these application methods are available through StateRover by leveraging an interface with the JUnit framework.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CASE STUDIES

In the previous chapter we explained how our process modeling approach can be applied to partially automated, human-based, certification and accreditation (C&A) processes. In this chapter, we provide two case studies to demonstrate the application of our statechart-based process modeling approach. These demonstrative exemplars show the utility of our approach.

The first case study examines the Cross Domain Implementation Process (CDIP) and the second examines the Cross Domain Solutions (CDS) Workflow. Early on in our research, the CDIP was being developed by the Unified Cross Domain Management Office (UCDMO) as the next-generation process for requesting, developing, implementing, and certifying and accrediting a cross domain solution. We applied our process modeling approach to the CDIP. This effort helped refine our approach.

During the course of our research, the UCDMO transitioned from the CDIP to the CDS Workflow process. These processes are related in that the CDS Workflow process is an evolved form of the CDIP. When UCDMO transitioned from the CDIP to the CDS Workflow as the process responsible for governing the request, development, implementation and C&A of cross domain solutions, we began to model the CDS Workflow process as well. This effort provided us with a number of benefits: application of our statechart-based modeling approach to two separate processes; fully exercising the runtime monitoring capabilities of the modeling approach; and validating the ability to apply process requirements as embedded assertions and enforce those assertions on an executing model of the process.

The statechart assertions shown in this chapter reflect the typical hard-to-model aspects of requirements on human-based processes. We demonstrate the modeling and V&V of several requirements for each modeled process. Formalizing all possible requirements on each process would not demonstrate anything additional and is recommended as future work in Section V.B.3 of this document. A full-scale implementation of all process requirements would include approximately 120 embedded statechart assertions for each modeled process.

We believe the CDIP and CDS Workflow process are particularly well suited for use in experimenting with our approach, given that these processes involve human decision making, temporal constraints and restraints, nested sub-processes, workflow elements and state changes.

A. CROSS DOMAIN IMPLEMENTATION PROCESS

1. Process Selection

The primary mission of the UCDMO is to support the timely delivery of secure, robust, and cost-effective cross domain capabilities and enterprise services that enable authorized US Government and strategic partner communities to safely share information across security domains (Unified Cross Domain Management Office 2012). The UCDMO has several concurrent initiatives designed to align and federate the implementation and support of cross domain solutions (CDS). The UCDMO recently published the following guidance materials on cross domain (CD) implementations of information systems—CD Community Roadmap, CD Inventory List, and CD Implementation Process (CDIP), all of which are available at the UCDMO Intelink website (Unified Cross Domain Management Office 2012). In this section we show the results of applying our modeling approach to the largely human-based CDIP (see Figure 27), which demonstrates the use of formal methods to specify and reason about a process designed to implement cross domain solutions.

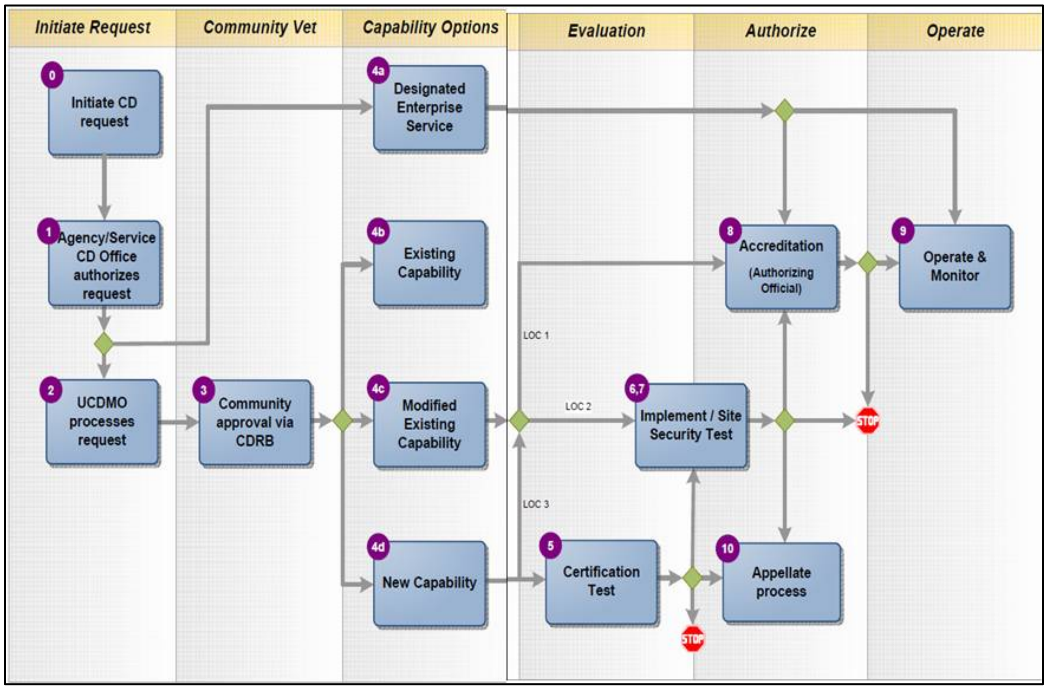

Initiate Request
Community Vet
Capability Options
Evaluation
Authorize
Operate
0 Initiate CD request
1 Agency/Service CD Office authorizes request
2 UCDMO processes request
3 Community approval via CDRB
4a Designated Enterprise Service
4b Existing Capability
4c Modified Existing Capability
4d New Capability
5 Certification Test
6,7 Implement / Site Security Test
8 Accreditation (Authorizing Official)
9 Operate & Monitor
10 Appellate process
LOC 1
LOC 2
LOC 3
STOP
STOP

Figure 27. Cross Domain Implementation Process Informal Diagram

Encompassed within the CDIP is the Intelligence Community Directive 503 (ICD 503) C&A process (Director of National Intelligence 2008). This process is the means by which the designated authorities such as the Cross Domain Resolution Board (CDRB) decide whether to allow a given CDS to operate. The UCDMO is not a decision making body; rather, it is responsible for the development, coordination, and oversight of the CDIP and its successor, the CDS Workflow process. We view the process for developing, implementing, certifying and accrediting cross domain solutions as critical to building the evidence necessary for decision makers to weigh the risks of operating a given CDS and to make the accreditation decision for the system.

2. Process Analysis

In the second step on the road to building a formal, statechart-based process model, thorough analysis helps us develop a better understanding of the process under examination. This analysis begins by gathering all available information related to the process. In the case of the CDIP, the process was still in the formative stages and the available documentation consisted of the informal flowchart shown in Figure 27 and several conceptual PowerPoint presentations attributable to members of the UCDMO team responsible for developing the CDIP (Unified Cross Domain Management Office 2008).

The CDIP is designed as a process that is easy for humans to understand and follow. Historically, the field of formal methods was born out of a need to rigorously specify and then perform verification and validation on systems, especially in the case of security- and safety-critical systems (Monin and Hinchey 2003). Formal methods tools and techniques are based on mathematical theories. One of the challenges of formally modeling a process designed for humans is capturing those portions of the process that involve subjective human activities like evaluation and decision making. For example, Step 1 “Authorize Request” of the CDIP demonstrates the subjective nature of the process (see Figure 28). In this step, a newly initiated cross domain request form (CDRF) must be validated and authorized or rejected by a human within the requestor’s agency/service CD office—a human-centric activity. Such activities need to be formally specified within the context of the process and we do so using the diamond-shaped visual switch artifact shown in Figure 28.

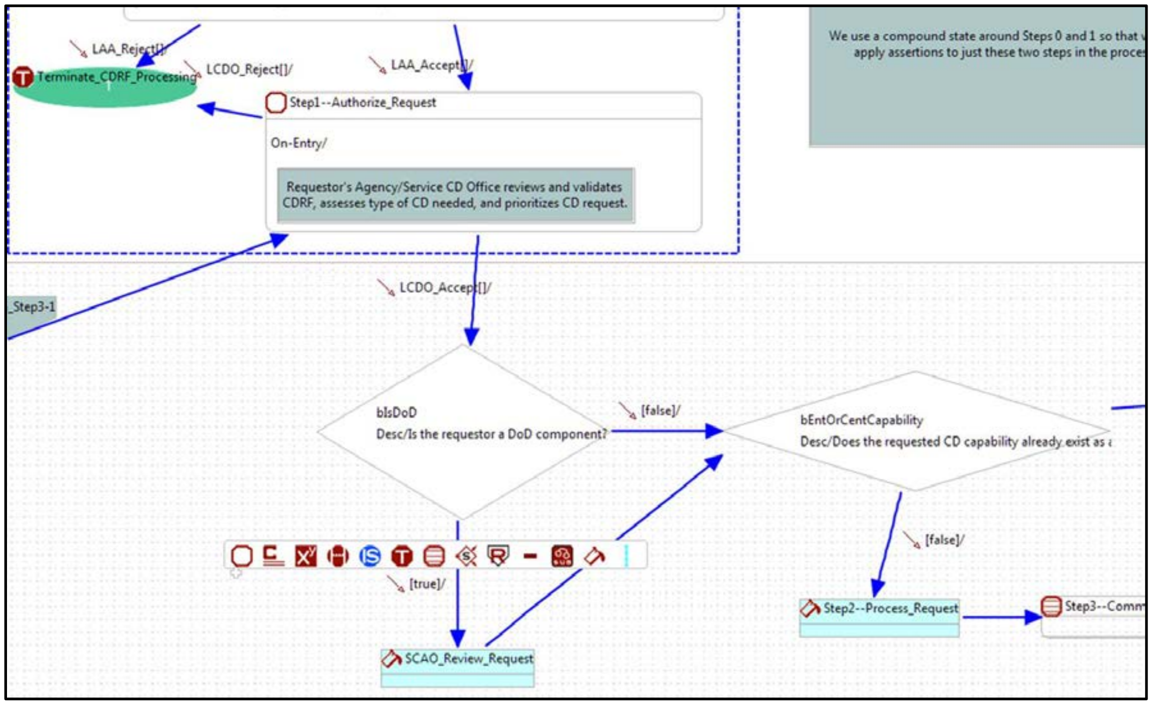

LAA_Reject[]/
Terminate_CDRF_Processing
LCDO_Reject[]/
LAA_Accept[]/
Step1--Authorize_Request
On-Entry/
Requestor's Agency/Service CD Office reviews and validates CDRF, assesses type of CD needed, and prioritizes CD request.
We use a compound state around Steps 0 and 1 so that
apply assertions to just these two steps in the proce
_Step3-1
LCDO_Accept[]/
bIsDoD
Desc/Is the requestor a DoD component?
[false]/
bEntOrCentCapability
Desc/Does the requested CD capability already exist as
[false]/
[true]/
Step2--Process_Request
Step3--Comm
SCAO_Review_Request

Figure 28. CDIP Step 1 “Authorize Request”

Our analysis revealed that the CDIP is a more complicated process than the flowchart of Figure 27 makes it appear. In fact, after evaluating the available documentation, we found the CDIP to be a complex, multi-threaded, multi-layered, temporally constrained process.

As prescribed by our process modeling approach, we identified individual threads, decision points, and key elements for translation to the type of statechart artifacts used when building StateRover models (e.g., states, visual switches). Of note, the list of identified threads only includes those inherent to the process; we do not include instances of threads used as an enabler for embedded assertions.

a. Threads

During *Step 5 – Certification Test*, a system is laboratory tested for compliance with mandated security requirements. If a system passes this testing phase it then moves to *Step 6 – Implement*. At the same time as a system proceeds to *Step 6 – Implement* it is evaluated as to whether it should be included in the CD Baseline Systems list. This evaluation proceeds independent of and concurrent with the system's implementation and subsequent site testing and therefore fits our criteria for identification as a thread.

b. Decision Points

As described in Section III.C.1 we use Table 5 to manage and track decision points for inclusion in the process model.

Table 5. CDIP Decision Points

Description	**Variable Name**	**Possible Values**	**In Model**
Decide whether CDS requestor is a DoD component.	bIsDod	True, false	☒
Determine whether capability exists as an enterprise or centralized capability.	bEntOrCentCapaiblity	True, false	☒
Decide how to transition a CDS based on the associated level of change	iClassOfChange	1, 2, 3	☒
Decide whether to initiate an appeal if CDS request is denied.	bInitAppeal	True, false	☒
Should CD request form (CDRF) move forward through process after CDRB review?	bCDRF_Proceed	True, false	☒
Does CDRF need to be revalidated after CDRB review?	bCDRF_Revalidate	True, false	☒
Does the CD capability already exist in a fully implemented version?	bCapabilityExists	True, false	☒
Can an existing CD capability be modified to meet the requirement?	bModifyCapability	True, false	☒
Is a completely new CDS required in order to meet the requested need?	bNewCapability	True, false	☒
Based on results of laboratory security test, decide whether to move CDS to next step of CDIP.	bProceedFwd	True, false	☒
Decide whether to add CDS as a baseline system.	bAddToBaseline	True, false	☒
Decide how to proceed based on results of ST&E.	bSTE_Successful	True, false	☒
What is the result of the accreditation process?	bAccredCD	True, false	☒

c. Layers

The CDIP is composed of processes and sub-processes. Therefore, we used statechart hierarchy when building the model. We determined that two levels of hierarchy are required in order to accurately capture the nested processes within the CDIP. In the diagram of Figure 27, each of *Step 3 – Community Approval Via CDRB*, *Step 5 – Certification Test*, *Step 6/7 – Implement / Site Security Testing*, and *Step 8 - Accreditation* were of sufficient complexity to warrant individual articulation as a nested process.

3. Construct Process Model

In this section, we bring together the results of process selection and analysis to articulate the formal process model of the CDIP. The diagrams of Figure 29 and Figure 30 show the right and left halves of the top-level view of the full process model. We have split this diagram into two figures for clarity.

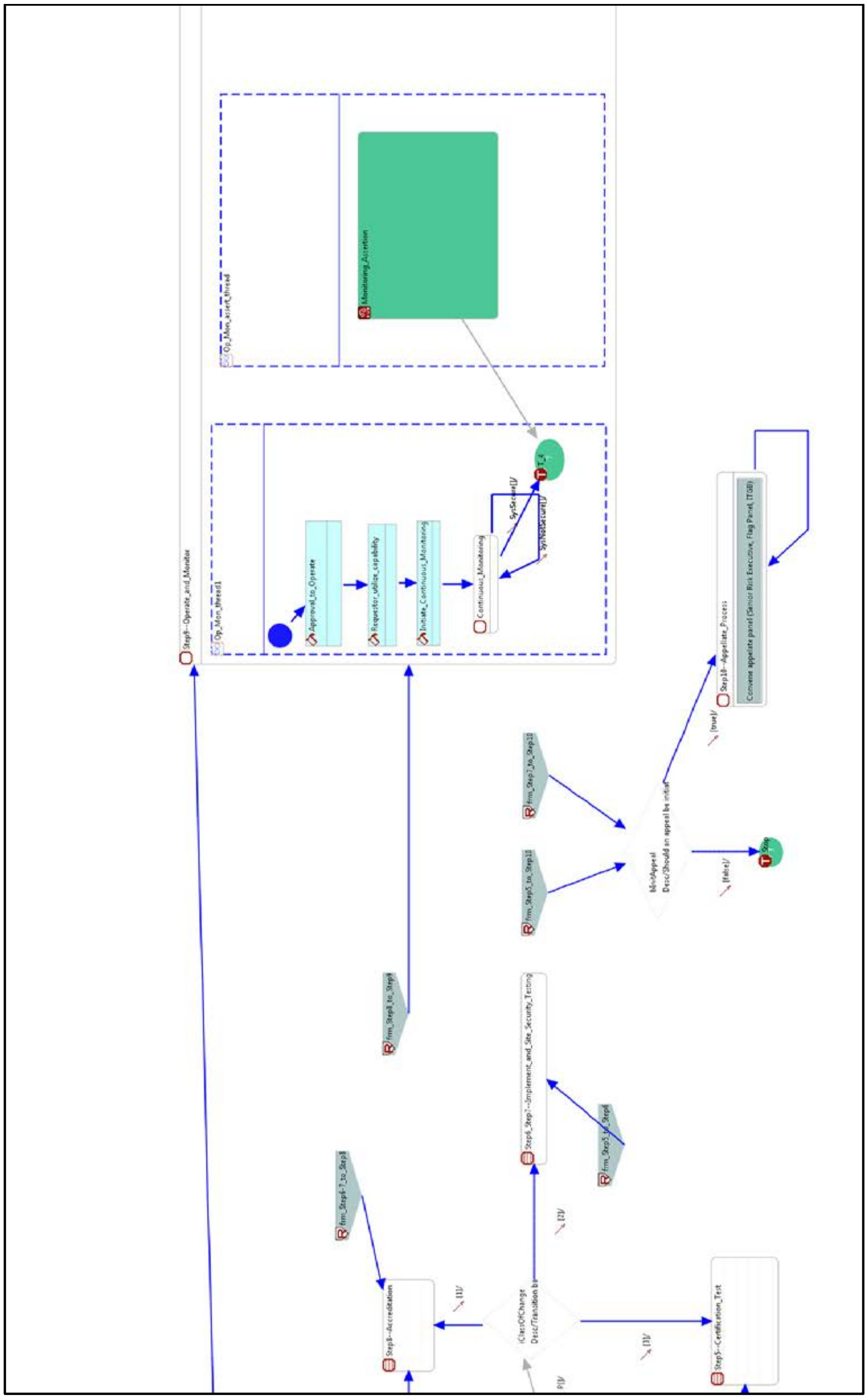
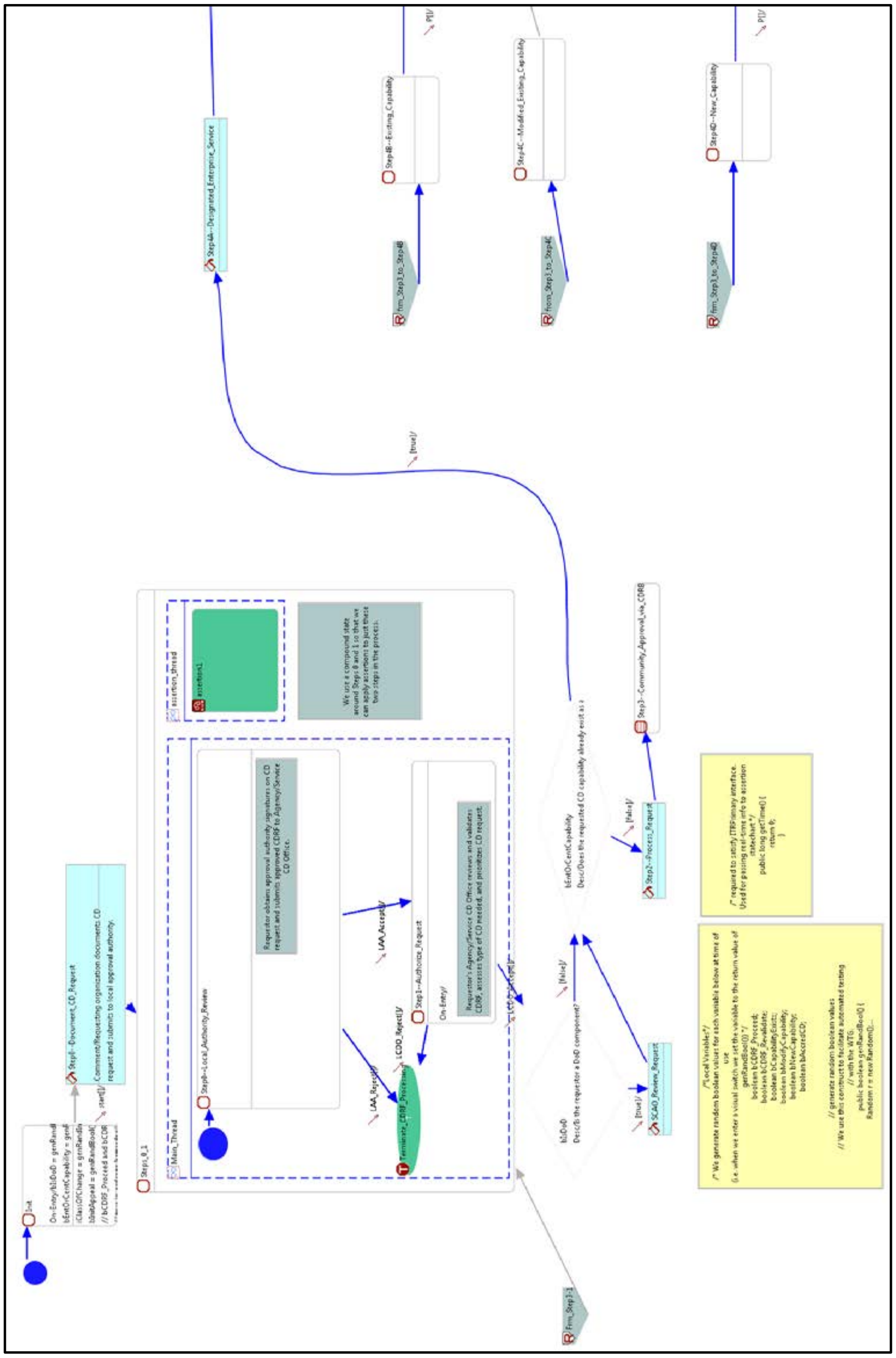


Figure 29. Top-Level View of Final CDIP Process Model (Right Half)



Init
Step0--Document_CD_Request
Steps_0_1
Main_Thread
Step0--Local_Authority_Review
Step1--Authorize_Request
Terminate_CDIP_Processing
assertion_thread
Step2--Process_Request
SCAO_Review_Request
Step3--Community_Approval_via_CDRB
Step4A--Designated_Enterprise_Service
Step4B--Existing_Capability
Step4C--Modified_Existing_Capability
Step4D--New_Capability
[true]/
[false]/

Figure 30. Top-Level View of Final CDIP Process Model (Left Half)

However, model development was an iterative process. By taking advantage of our chosen modeling tool's code generation capability and inherent semantic compliance checking routines, we were able to use a build-and-check approach to iteratively constructing the model. The CDIP is a stepwise design, lending itself to building and testing in sections. Throughout model development and with the addition of each new step we take two actions designed to ensure the model adheres to the underlying rules for semantic correctness: (i) Use the "Diagram/Validate" menu option to initiate StateRover's validation routine and reveal detected errors and (ii) initiate the code generation process. As detailed in Chapter III, diagram validation and code generation provide an end-to-end syntactic and semantic check of the model and identify errors. Multiple types of errors could be detected through this process such as mistakes made by the stakeholders in the formulation of the natural language representations of the process and its requirements which get built into the model by the process engineer, errors by the process engineer when translating the natural language into the model and its assertions, or mistakes by the process engineer when creating the model (e.g., sink states, loops).

As shown in Figure 31, we initially constructed Steps 0, 1, 2, and a placeholder coarse-state for Step 3 (outlined in green). The visual switch transition *[true]* from *bEntOrCentCapability* to *Step2 – Process_Request* will go to *Step 4A – Designate_Enterprise_Service* in the final version of the model. However, in order to pass the syntactic check for both true and false transitions from a visual switch we temporarily route the *[true]* transition (outlined in red) to *Step2 – Process_Request* as shown in Figure 31. On the right hand side of state *Steps_0_1* we positioned a placeholder thread, named *assertion_thread*, which will contain the embedded assertion to be placed later in the modeling process.

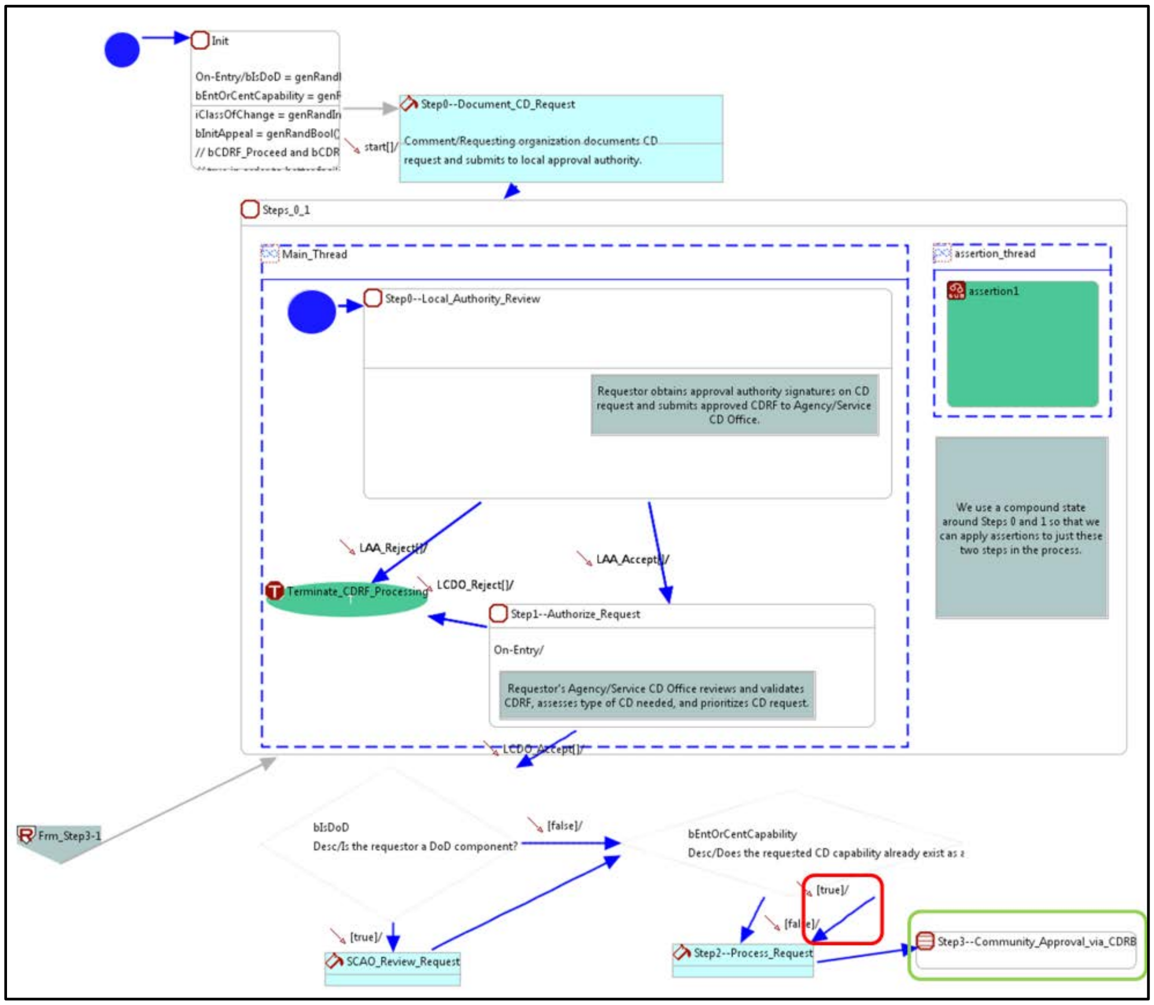

Init
On-Entry/bIsDoD = genRandI
bEntOrCentCapability = genF
iClassOfChange = genRandIn
bInitAppeal = genRandBool(
// bCDRF_Proceed and bCDR
start[]/
Step0--Document_CD_Request
Comment/Requesting organization documents CD request and submits to local approval authority.
Steps_0_1
Main_Thread
assertion_thread
assertion1
Step0--Local_Authority_Review
Requestor obtains approval authority signatures on CD request and submits approved CDRF to Agency/Service CD Office.
We use a compound state around Steps 0 and 1 so that we can apply assertions to just these two steps in the process.
LAA_Reject[]/
LAA_Accept[]/
Terminate_CDRF_Processing
LCDO_Reject[]/
Step1--Authorize_Request
On-Entry/
Requestor's Agency/Service CD Office reviews and validates CDRF, assesses type of CD needed, and prioritizes CD request.
LCDO_Accept[]/
Frm_Step3-1
bIsDoD
Desc/Is the requestor a DoD component?
[false]/
bEntOrCentCapability
Desc/Does the requested CD capability already exist as a
[true]/
[false]/
[true]/
SCAO_Review_Request
Step2--Process_Request
Step3--Community_Approval_via_CDRB

Figure 31. CDIP Steps 0 - 3 Top-level View During Model Development

Next, we constructed the detailed view of *Step 3* (Figure 32). The capability to display *Step 3*'s single coarse-state placeholder on the diagram of Figure 31 and expand that view as shown in Figure 32, demonstrates one of the benefits of hierarchy, that is, adjusting the depth and complexity of the view as needed. *Step 3* is characterized by a number of questions that must be answered in order to properly route a CDS request into the appropriate path for the next step of the process.

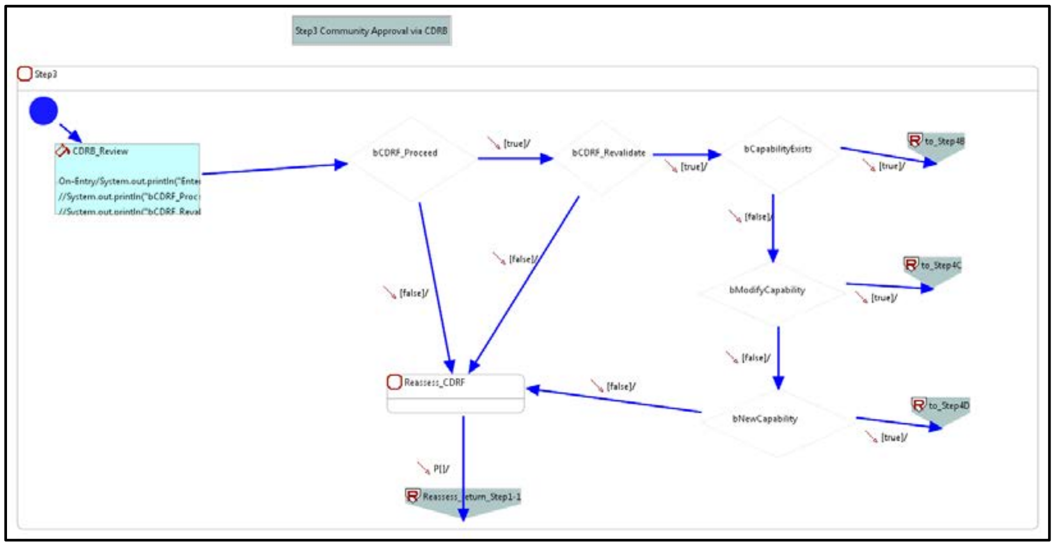


Figure 32. Step 3–Community_Approval_Via_CDRB

The three visual switches on the right hand side of Figure 32 represent human decisions that determine whether: (i) the requested capability already exist within the inventory; (ii) an existing capability can be modified to meet the stated requirement; or (iii) authority and funding for development of a new capability is approved in order to meet the requirements of the CDS request. Depending on the answer to each of these questions (i.e., true or false) the process will transition to the appropriate section of *Step 4* as designated on the far right hand side of Figure 32 by the off-page references *to_Step4B*, *to_Step4C*, and *to_Step4D*. If it is determined that there is no way to proceed forward with developing or acquiring the requested CDS capability then the process transitions back to the top-level statechart via *Reassess_return_Step_1-1*.

After constructing *Step_3* we once again run the validation and code generation routines to iteratively ensure the model is semantically and syntactically correct. We do this at each stage of model development.

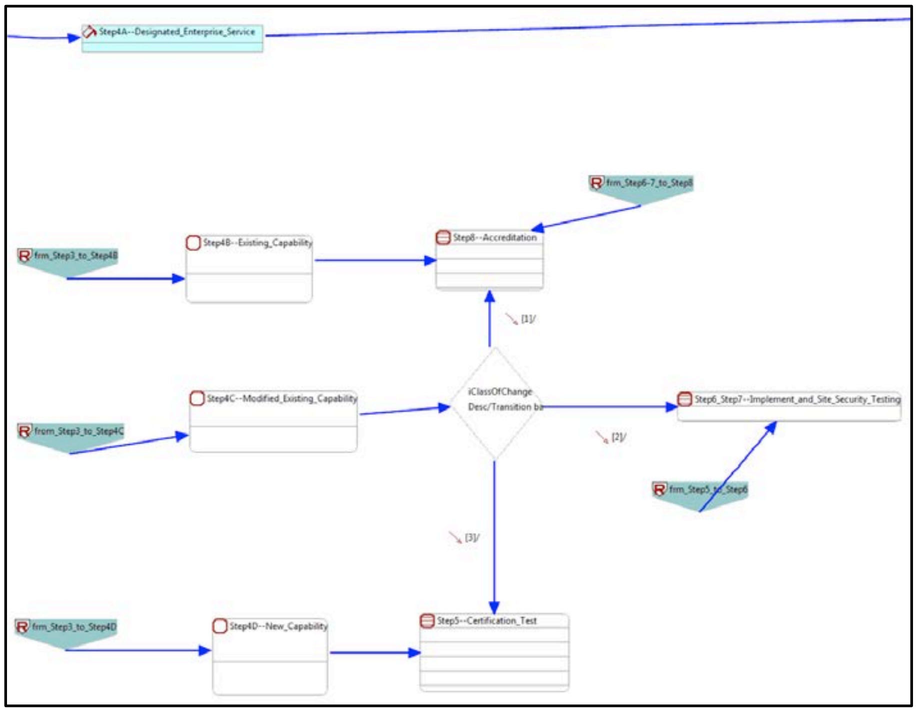
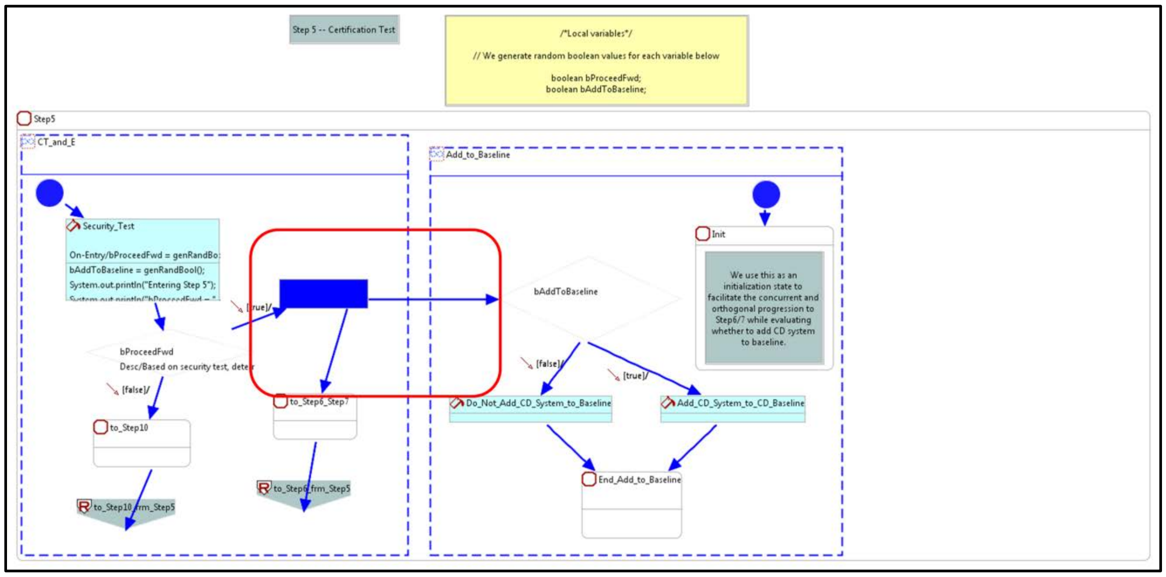


Figure 33. CDIP Top-level Development of Steps 4, 5, and 6/7

The top-level view of *Step 4(a, b, c, d)*, *Step 5*, and *Step 6/7* is shown in Figure 33. Again we use coarse states for those steps that will be further articulated as sub-processes *Step 5, Step 6/7,* and *Step 8*. The detailed view for each of these steps is shown in Figure 34, Figure 35, and Figure 36, respectively.



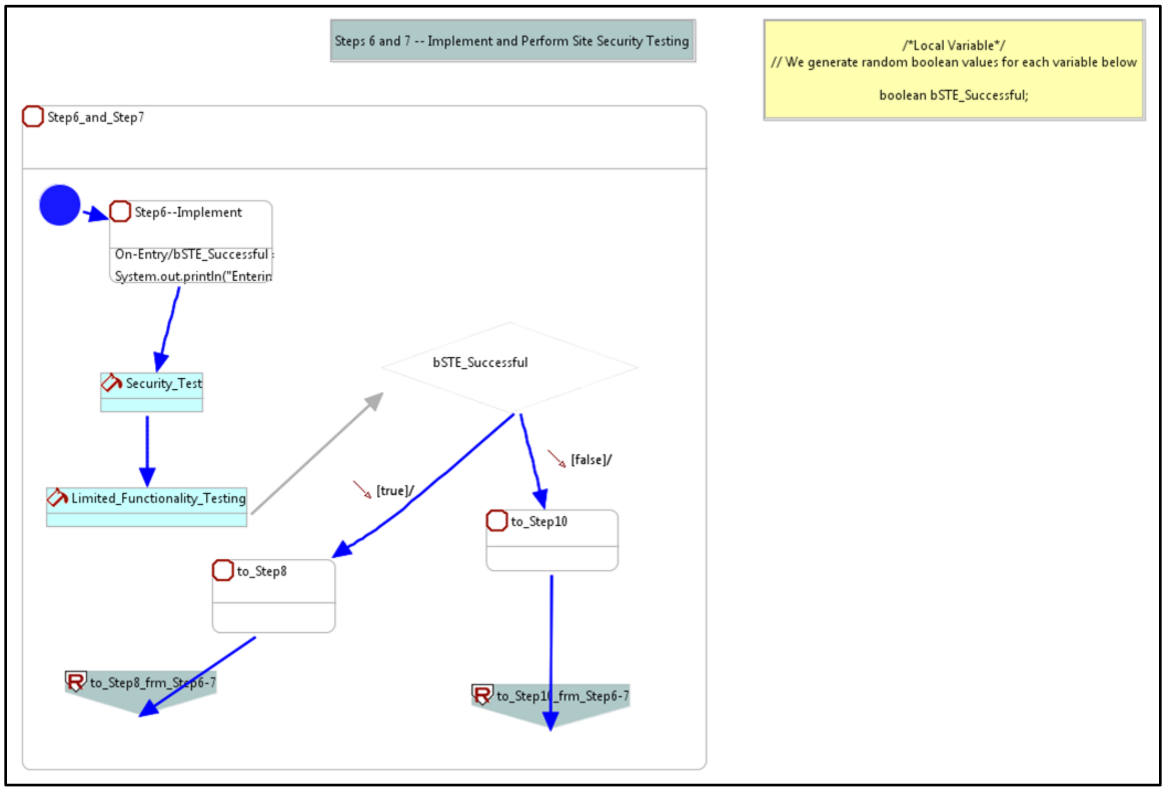
```
/*Local variables*/

// We generate random boolean values for each variable below

boolean bProceedFwd;
boolean bAddToBaseline;
```

```
On-Entry/bProceedFwd = genRandBo
bAddToBaseline = genRandBool();
System.out.println("Entering Step 5");
```

We use this as an initialization state to facilitate the concurrent and orthogonal progression to Step6/7 while evaluating whether to add CD system to baseline.

Figure 34. Step 5 – Certification Test

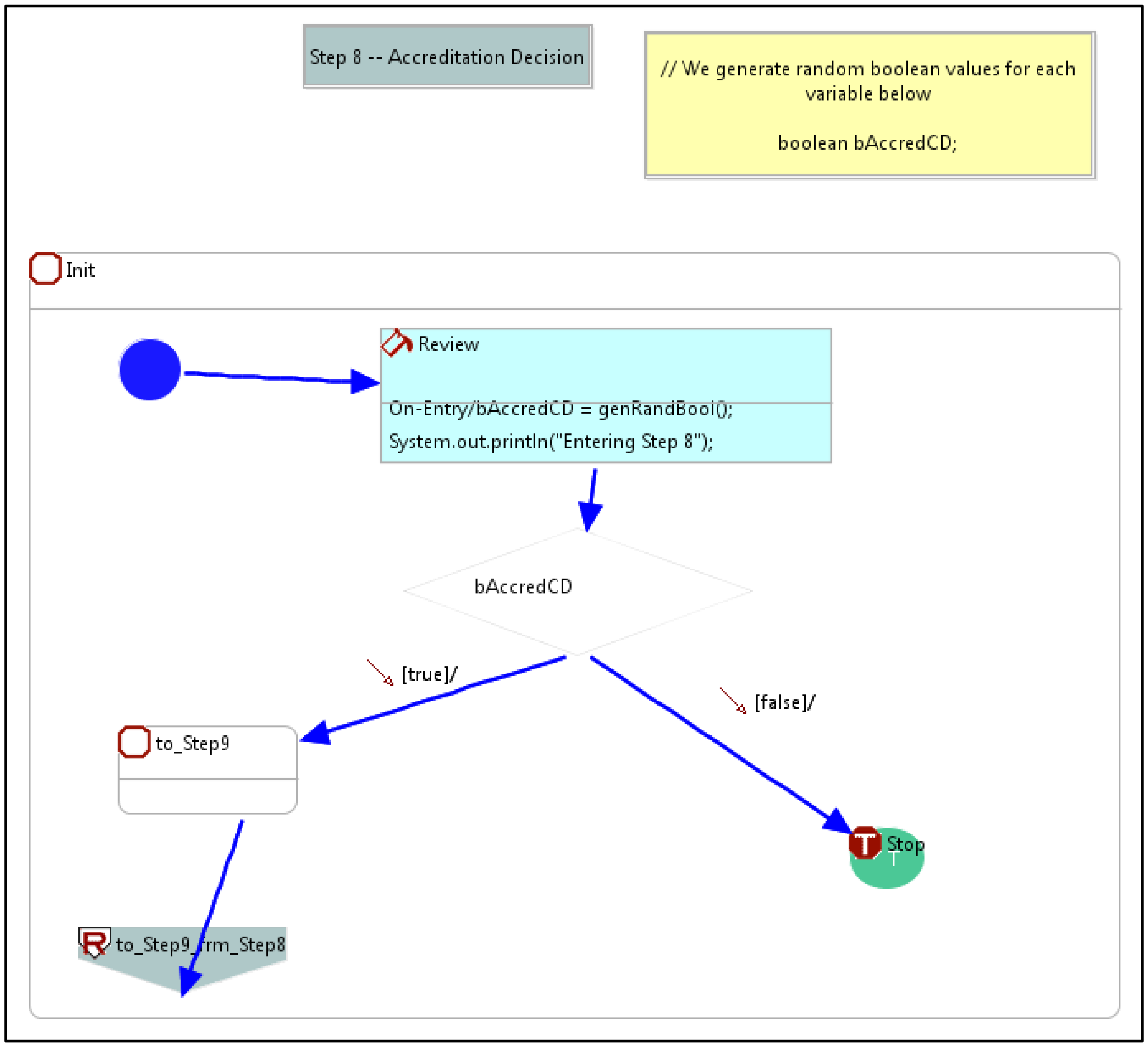
In *Step 5 – Certification Test* there are two statechart threads. This construct, in conjunction with the transition connector artifact (outlined in red in Figure 34) permits the process engineer to enable interaction and transitions between concurrent activities within a process. In this case, after security testing is complete and a decision has been made to proceed forward with implementing and site testing a CDS, the process also calls for a review of the system to decide if it should be added to the CDS baseline.



```
/*Local Variable*/
// We generate random boolean values for each variable below

boolean bSTE_Successful;
```

```
On-Entry/bSTE_Successful
System.out.println("Enterin
```

Figure 35. Step 6/7 – Implement and Perform Site Security Testing



```
// We generate random boolean values for each
variable below

boolean bAccredCD;
```

```
On-Entry/bAccredCD = genRandBool();
System.out.println("Entering Step 8");
```

Figure 36. Step 8 – Accreditation Decision

After the construction of each step we validate and generate code to ensure that the model is error-free. If the model is not error-free, the process engineer makes corrections as required and then validates the model and generates code again. This process continues until each new section of the model is error-free.

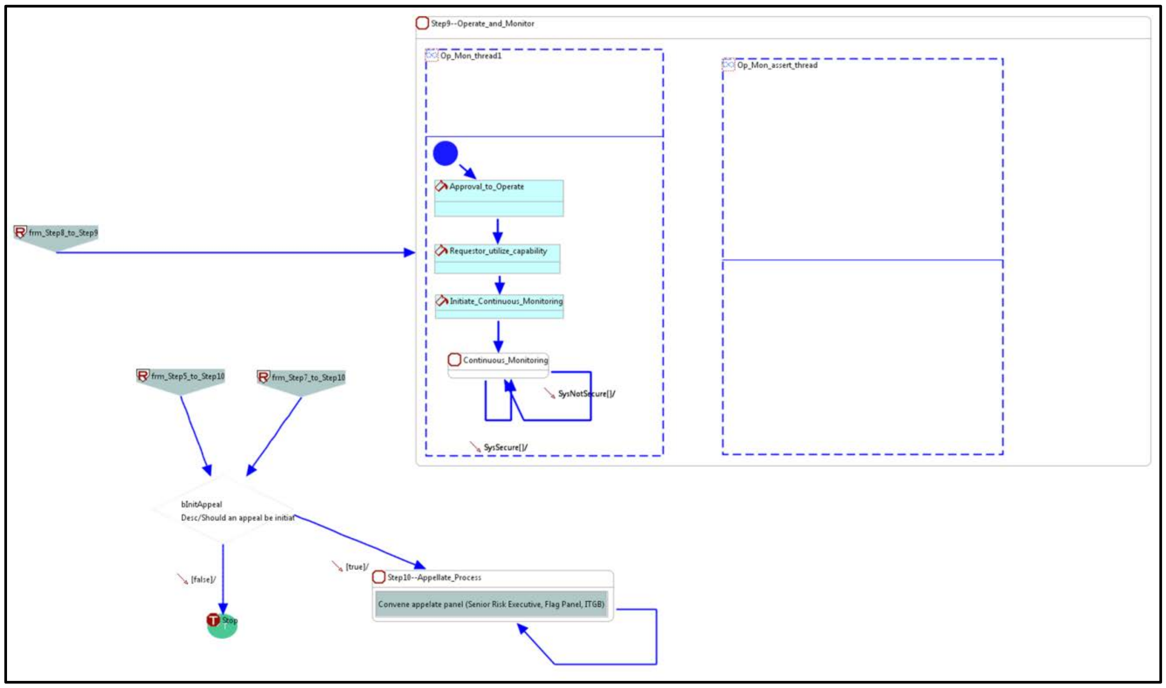

Step9--Operate_and_Monitor
Op_Mon_thread1
Op_Mon_assert_thread
frm_Step8_to_Step9
Approval_to_Operate
Requestor_utilize_capability
Initiate_Continuous_Monitoring
Continuous_Monitoring
SysNotSecure[]/
SysSecure[]/
frm_Step5_to_Step10
frm_Step7_to_Step10
bInitAppeal
Desc/Should an appeal be initiat
[false]/
[true]/
Stop
Step10--Appellate_Process
Convene appelate panel (Senior Risk Executive, Flag Panel, ITGB)

Figure 37. CDIP Top-level View *Steps 9, 10*

Next, we constructed the final portion of the top-level model, including *Step 9* and *Step 10* (see Figure 37). From a process engineering perspective, we made a design decision to place *Step 9* in the top-level of the model. This helps demonstrate the flexibility and scalability of the modeling approach. If placing *Steps 9* and *10* in the top-level had significantly increased the difficulty of understanding the model or following its flow then we would have been able to represent it as a coarse state at the top-level and fully articulate it in a separate substatechart as we did for *Steps 3*, *5*, *6/7*, and *8*. On the right hand side of state *Step 9 – Operate_and_Monitor* we positioned a placeholder thread, named *Op_Mon_assert_thread* that will contain the embedded assertion to be placed later in the modeling process.

4. Construct Statechart Assertions

In this section we show two embedded statechart assertions, each designed to model and enforce a different natural language requirement. This demonstrates the technical feasibility of applying statechart embedded assertions to the type of partially automated human-based C&A processes modeled with our approach.

The statechart assertion of Figure 38 is a formal specification of the natural language requirement R1: *Local approval authority/authorities must ensure that there is a valid operational need for CDS.*

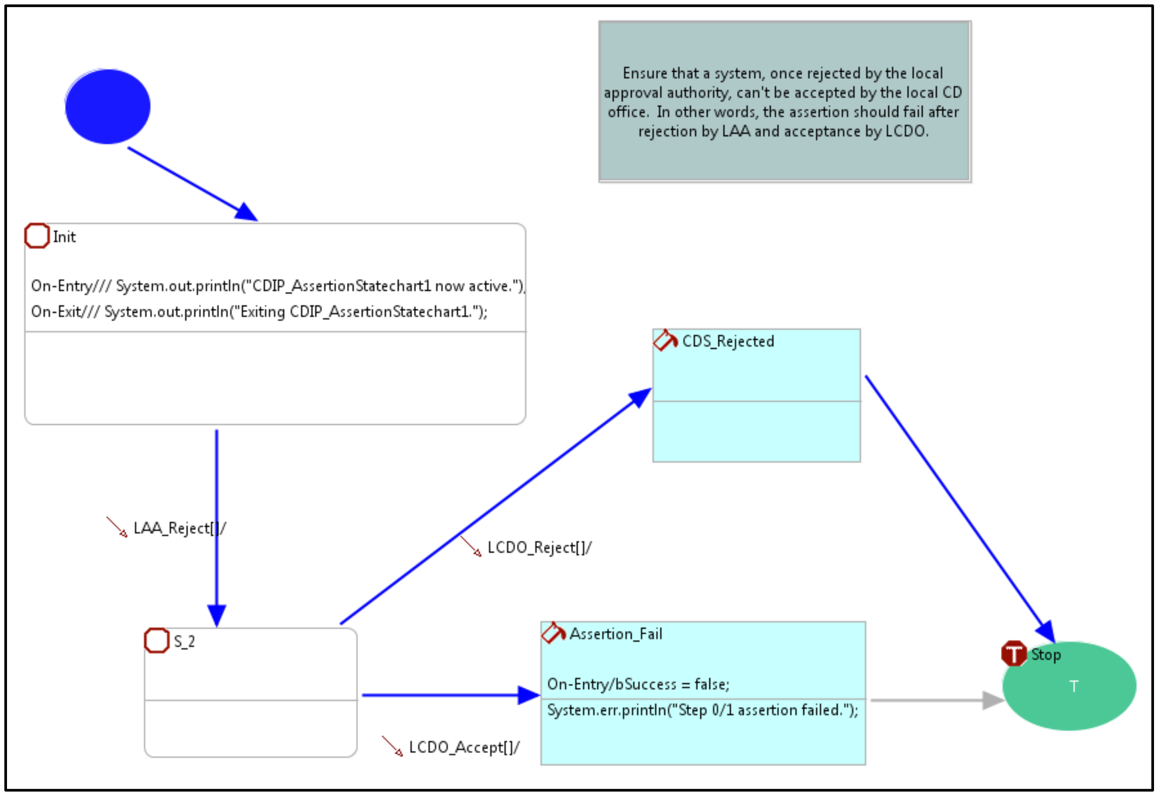

Ensure that a system, once rejected by the local approval authority, can't be accepted by the local CD office. In other words, the assertion should fail after rejection by LAA and acceptance by LCDO.
Init
On-Entry/// System.out.println("CDIP_AssertionStatechart1 now active.");
On-Exit/// System.out.println("Exiting CDIP_AssertionStatechart1.");
CDS_Rejected
LAA_Reject[]/
LCDO_Reject[]/
S_2
Assertion_Fail
On-Entry/bSuccess = false;
System.err.println("Step 0/1 assertion failed.");
Stop
T
LCDO_Accept[]/

Figure 38. Statechart Assertion for Requirement R1

The statechart assertion of Figure 39 is a formal specification of the natural language requirement R2: *the CDS implementation must remain secure during the "operate and monitor" phase of its lifecycle*.

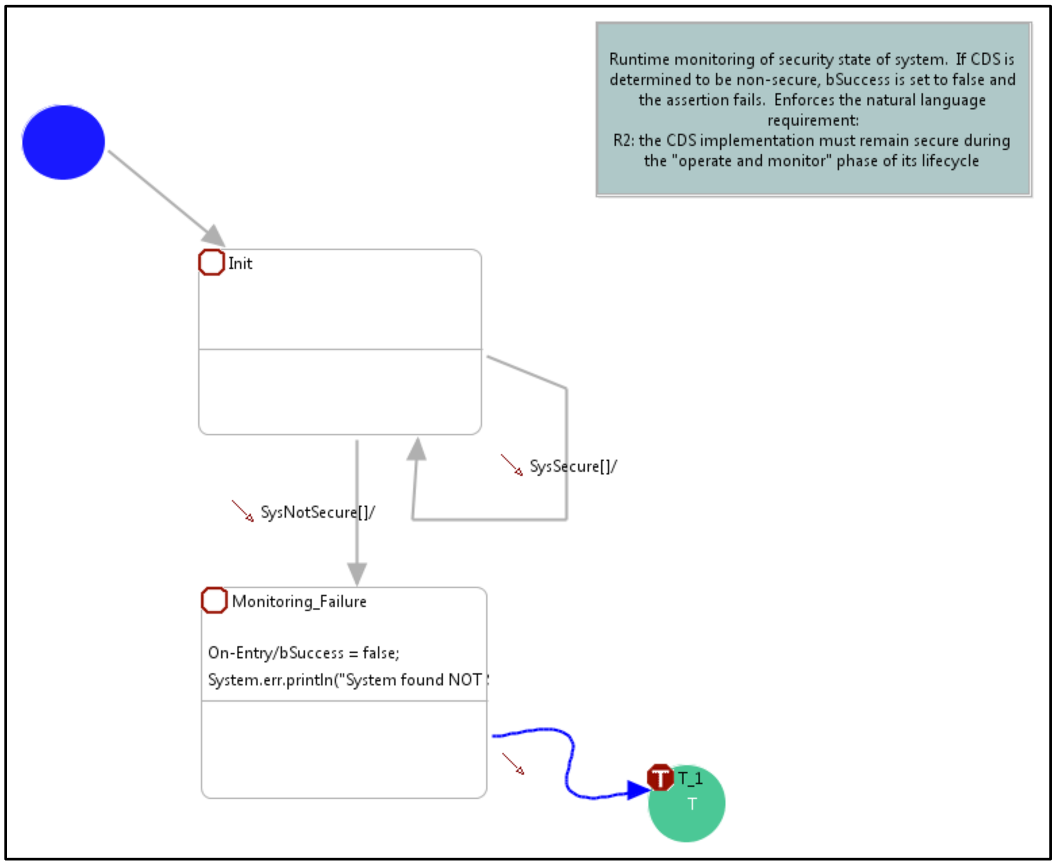

Runtime monitoring of security state of system. If CDS is determined to be non-secure, bSuccess is set to false and the assertion fails. Enforces the natural language requirement:
R2: the CDS implementation must remain secure during the "operate and monitor" phase of its lifecycle
Init
SysSecure[]/
SysNotSecure[]/
Monitoring_Failure
On-Entry/bSuccess = false;
System.err.println("System found NOT
T_1
T

Figure 39. Statechart Assertion for Requirement R2

a. Validating Statechart Assertions

As described in Section III.C.4.b, we use a pattern-based testing methodology to help build a body of evidence that our assertions correctly represent the intended behaviors. Figure 40 shows an example of a testing scenario for the assertion of Figure 38 to test the *obvious success* pattern.

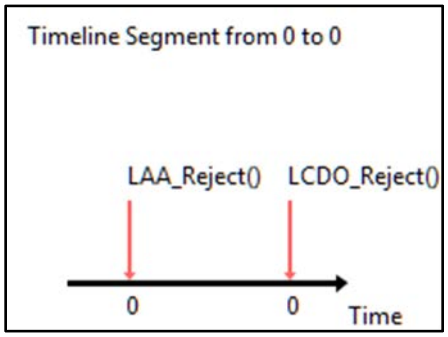


Figure 40. Timeline Diagram for "*obvious success*" Assertion Test Scenario

This is a trivial case but important not to overlook. Using this methodology, on several occasions throughout the development process we found the need to make corrections based on test results from pattern-based testing scenarios. For example, in the statechart assertion of Figure 38 we initially reversed the placement of the *LCDO_Reject()* and *LCDO_Accept()* events on their respective transitions. The trivial *obvious success* based test pattern revealed this error, demonstrating the value of exposing process models to a wide range of test scenarios from trivial to complex.

5. Embed Assertions in Process Model

In the next phase of our process modeling approach we embed the statechart assertions into the process model. As discussed in Section III.C.5 of this document, the process engineer places embedded statechart assertions within the model based on the desired scope of the assertion.

The statechart assertion of Figure 41 is only applicable to activity modeled in state *Steps_0_1*; therefore, to keep it appropriately scoped we have embedded it in a thread via the substatechart artifact of our chosen modeling tool as shown in Figure 38.

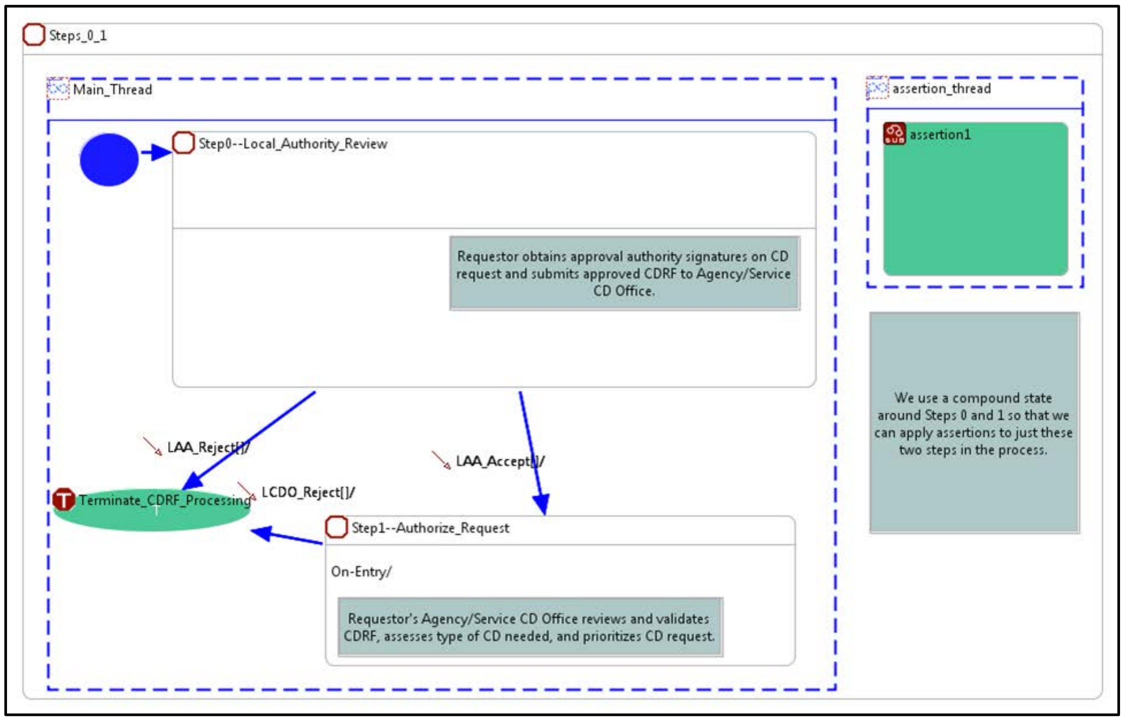

Steps_0_1
Main_Thread
Step0--Local_Authority_Review
Requestor obtains approval authority signatures on CD request and submits approved CDRF to Agency/Service CD Office.
LAA_Reject[]/
LAA_Accept[]/
LCDO_Reject[]/
Terminate_CDRF_Processing
Step1--Authorize_Request
On-Entry/
Requestor's Agency/Service CD Office reviews and validates CDRF, assesses type of CD needed, and prioritizes CD request.
assertion_thread
assertion1
We use a compound state around Steps 0 and 1 so that we can apply assertions to just these two steps in the process.

Figure 41. *Steps_0_1* with Embedded Statechart Assertion

The statechart assertion for R2, shown in Figure 39, is only applicable to the activities and events occurring in *Step 9* so it has been embedded within a thread (see Figure 42) of this state to maintain the appropriate scope.

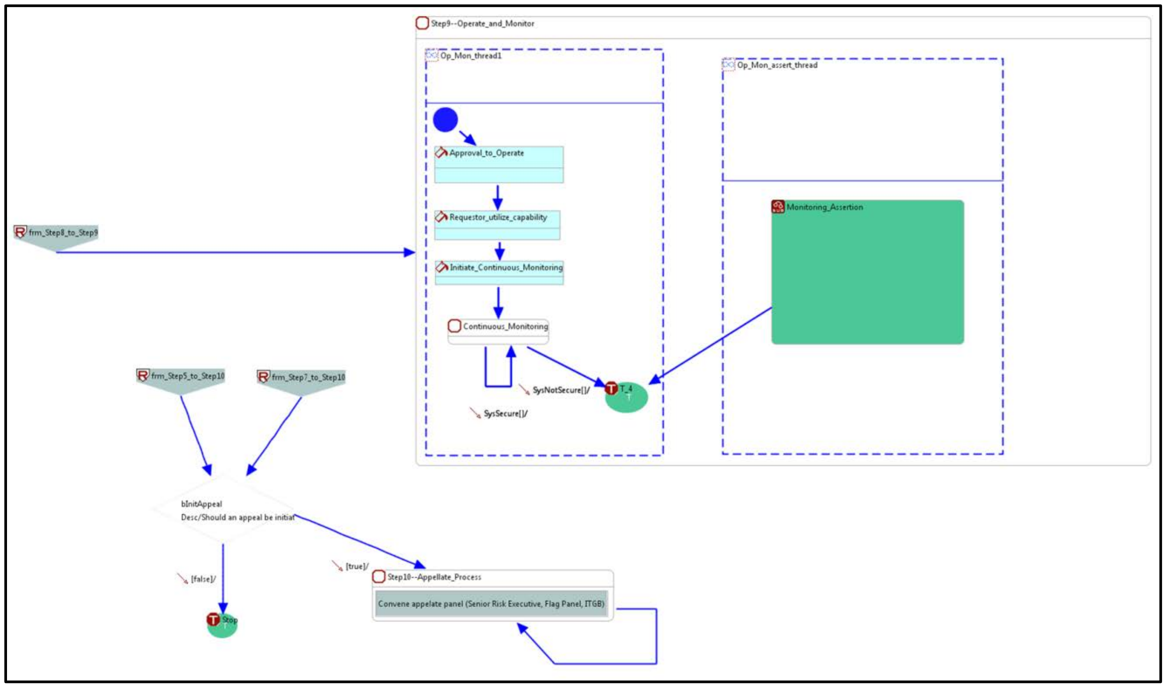


Figure 42. Step 9—Operate_and_Monitor – with Embedded Assertion

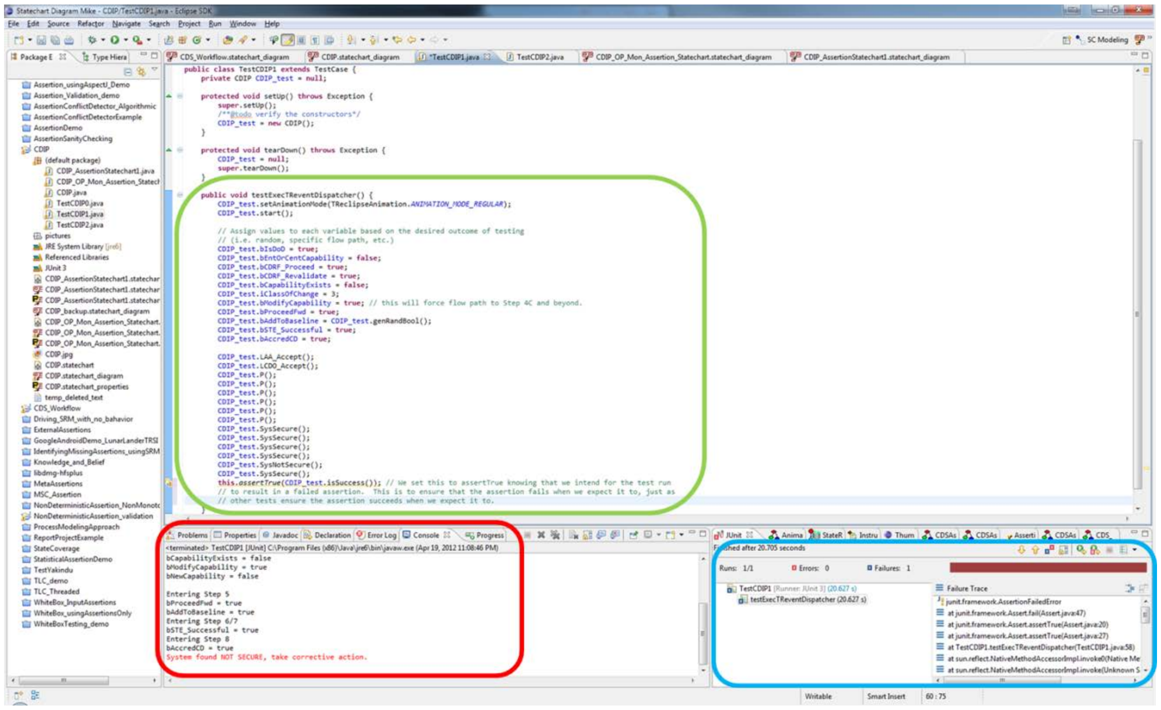
6. V&V Process Model

a. Validation

In the validation component of our modeling approach, the process engineer presents the finalized model to process stakeholders to ensure the model meets the expectations and requirements of stakeholders. This interaction is depicted in Figure 6 by the dashed line from *V&V Process* to *Process Stakeholder Expectations*. It represents a key portion of the feedback loop with stakeholders and facilitates validation of the process model. For the purposes of demonstrating the feasibility of this component of our modeling approach, we worked directly with the UCDMO to ensure the completed process model met their expectations. We presented diagrams of our formal model, like those presented in this chapter, and discussed the translation process from UCDMO supplied process documentation to the process model. Through informal discussion we asked questions about components of the model to ensure that we fully understood and had correctly translated stakeholder intent and requirements into the process model. The UCDMO personnel asked questions such as inquiring how the concurrent activities in *Step* 5 would be handled by our modeling approach and whether requirements enforcement through assertions could be designed into a long-term process monitoring system. Long-term process monitoring is an open research question that we propose as future work in Chapter V of this document.

b. Verification – Manual Testing

For manual testing, the process engineer writes Java-based test scripts that enact the scenarios he desires to test. Figure 43 shows a manually generated test case just after a test run with a failed assertion. The green outline contains commands used to initiate events and set variables in the process model being tested. Our chosen modeling software provides two types of feedback from the testing process. The red outline shows messages from the executing model and in this case shows the red status message, "System found NOT SECURE, take corrective action," which is generated by the statechart assertion CDIP*_OP_Mon_Assertion_Statechart* upon assertion failure. The blue outline shows JUnit reporting on failed assertions. This type of testing and the associated status messages from the model's embedded statechart assertions and JUnit demonstrate the means by which we are able to ensure process requirements, represented by statechart assertions, are enforced at runtime.



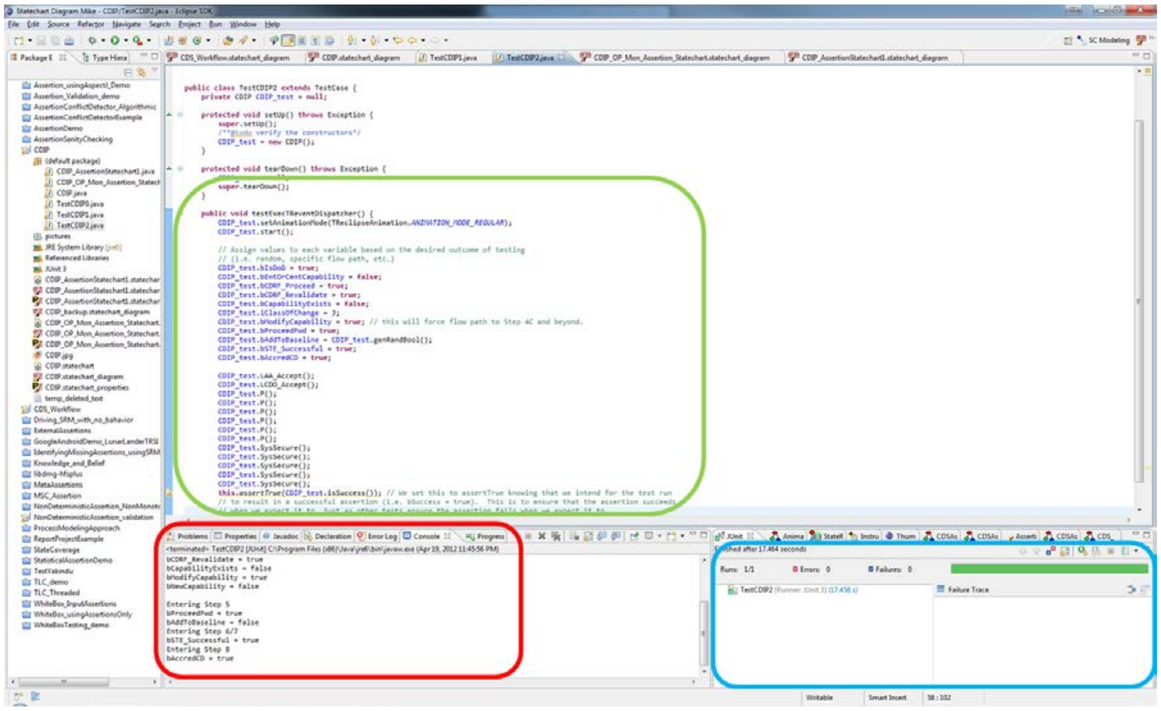
```java
public class TestCDIP1 extends TestCase {
    private CDIP CDIP_test = null;

    protected void setUp() throws Exception {
        super.setUp();
        /**@todo verify the constructors*/
        CDIP_test = new CDIP();
    }

    protected void tearDown() throws Exception {
        CDIP_test = null;
        super.tearDown();
    }

    public void testExecTReventDispatcher() {
        CDIP_test.setAnimationMode(TReclipseAnimation.ANIMATION_MODE_REGULAR);
        CDIP_test.start();

        // Assign values to each variable based on the desired outcome of testing
        // (i.e. random, specific flow path, etc.)
        CDIP_test.bIsDoD = true;
        CDIP_test.bEntOrCentCapability = false;
        CDIP_test.bCDRF_Proceed = true;
        CDIP_test.bCDRF_Revalidate = true;
        CDIP_test.bCapabilityExists = false;
        CDIP_test.iClassOfChange = 3;
        CDIP_test.bModifyCapability = true; // this will force flow path to Step 4C and beyond.
        CDIP_test.bProceedFwd = true;
        CDIP_test.bAddToBaseline = CDIP_test.genRandBool();
        CDIP_test.bSTE_Successful = true;
        CDIP_test.bAccredCD = true;

        CDIP_test.LAA_Accept();
        CDIP_test.LCDO_Accept();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        CDIP_test.SysNotSecure();
        CDIP_test.SysSecure();
        this.assertTrue(CDIP_test.isSuccess()); // We set this to assertTrue knowing that we intend for the test run
        // to result in a failed assertion.  This is to ensure that the assertion fails when we expect it to, just as
        // other tests ensure the assertion succeeds when we expect it to.
    }
```

```
<terminated> TestCDIP1 [JUnit] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Apr 19, 2012 11:08:46 PM)
bCapabilityExists = false
bModifyCapability = true
bNewCapability = false

Entering Step 5
bProceedFwd = true
bAddToBaseline = true
Entering Step 6/7
bSTE_Successful = true
Entering Step 8
bAccredCD = true
System found NOT SECURE, take corrective action.
```

```
Finished after 20.705 seconds
Runs: 1/1    Errors: 0    Failures: 1

TestCDIP1 [Runner: JUnit 3] (20.627 s)
    testExecTReventDispatcher (20.627 s)

Failure Trace
junit.framework.AssertionFailedError
at junit.framework.Assert.fail(Assert.java:47)
at junit.framework.Assert.assertTrue(Assert.java:20)
at junit.framework.Assert.assertTrue(Assert.java:27)
at TestCDIP1.testExecTReventDispatcher(TestCDIP1.java:58)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Me
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown S
```

Figure 43. Manual Testing Example – Failed Assertion

Figure 44 shows a similar test scenario but this one is set up to ensure the embedded assertion of Figure 39 succeeds when we expect it to. In this case, we have removed the line *CDIP_Test.SysNotSecure();* so the commands in the green outlined area do not drive the statechart assertion, CDIP*_OP_Mon_Assertion_Statechart*, to a failure condition. Since the assertion did not fail, the red outlined area does not show a failure message from the statechart assertion. The blue outlined area has a green bar, indicating that no assertions failed.



```
public class TestCDIP2 extends TestCase {
    private CDIP CDIP_test = null;

    protected void setUp() throws Exception {
        super.setUp();
        /**@todo verify the constructors*/
        CDIP_test = new CDIP();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testExecTReventDispatcher() {
        CDIP_test.setAnimationMode(TReclipseAnimation.ANIMATION_MODE_REGULAR);
        CDIP_test.start();

        // Assign values to each variable based on the desired outcome of testing
        // (i.e. random, specific flow path, etc.)
        CDIP_test.bIsDoD = true;
        CDIP_test.bEntOrCentCapability = false;
        CDIP_test.bCDRF_Proceed = true;
        CDIP_test.bCDRF_Revalidate = true;
        CDIP_test.bCapabilityExists = false;
        CDIP_test.iClassOfChange = 3;
        CDIP_test.bModifyCapability = true; // this will force flow path to Step 4C and beyond.
        CDIP_test.bProceedFwd = true;
        CDIP_test.bAddToBaseline = CDIP_test.genRandBool();
        CDIP_test.bSTE_Successful = true;
        CDIP_test.bAccredCD = true;

        CDIP_test.LAA_Accept();
        CDIP_test.LCDO_Accept();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.P();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        CDIP_test.SysSecure();
        this.assertTrue(CDIP_test.isSuccess()); // We set this to assertTrue knowing that we intend for the test run
        // to result in a successful assertion (i.e. bSuccess = true).  This is to ensure that the assertion succeeds
```

```
<terminated> TestCDIP2 [JUnit] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Apr 19, 2012 11:45:56 PM)
bCDRF_Revalidate = true
bCapabilityExists = false
bModifyCapability = true
bNewCapability = false

Entering Step 5
bProceedFwd = true
bAddToBaseline = false
Entering Step 6/7
bSTE_Successful = true
Entering Step 8
bAccredCD = true
```

```
Finished after 17.464 seconds
Runs: 1/1    Errors: 0    Failures: 0
TestCDIP2 [Runner: JUnit 3] (17.456 s)
```

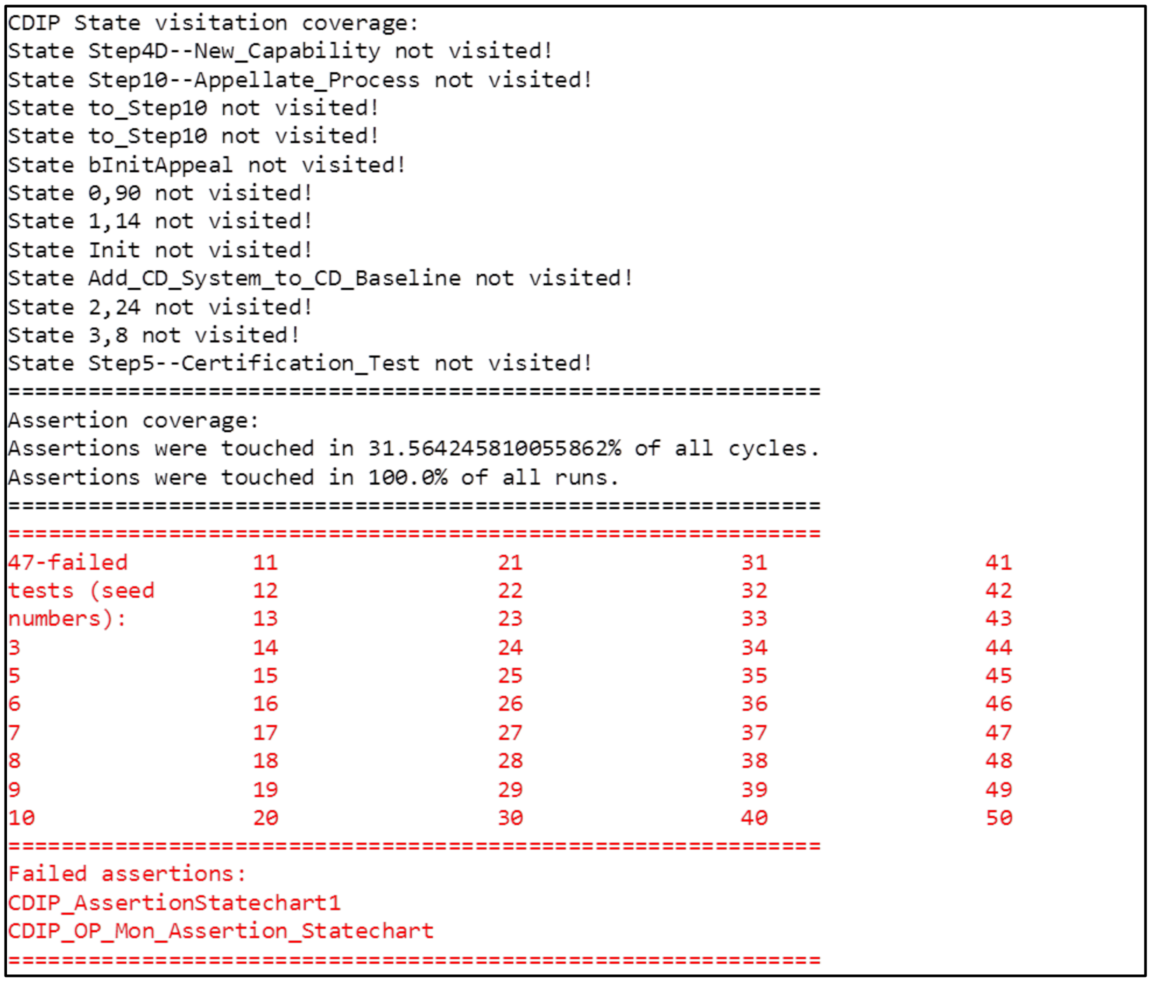
Figure 44. Manual Testing Example – Successful Assertion

The test cases we have shown demonstrate one of the methods by which we ensure that a process model behaves exactly as expected under specific conditions. The other method is through automated testing via runtime execution monitoring.

c. *Verification – Runtime Execution Monitoring*

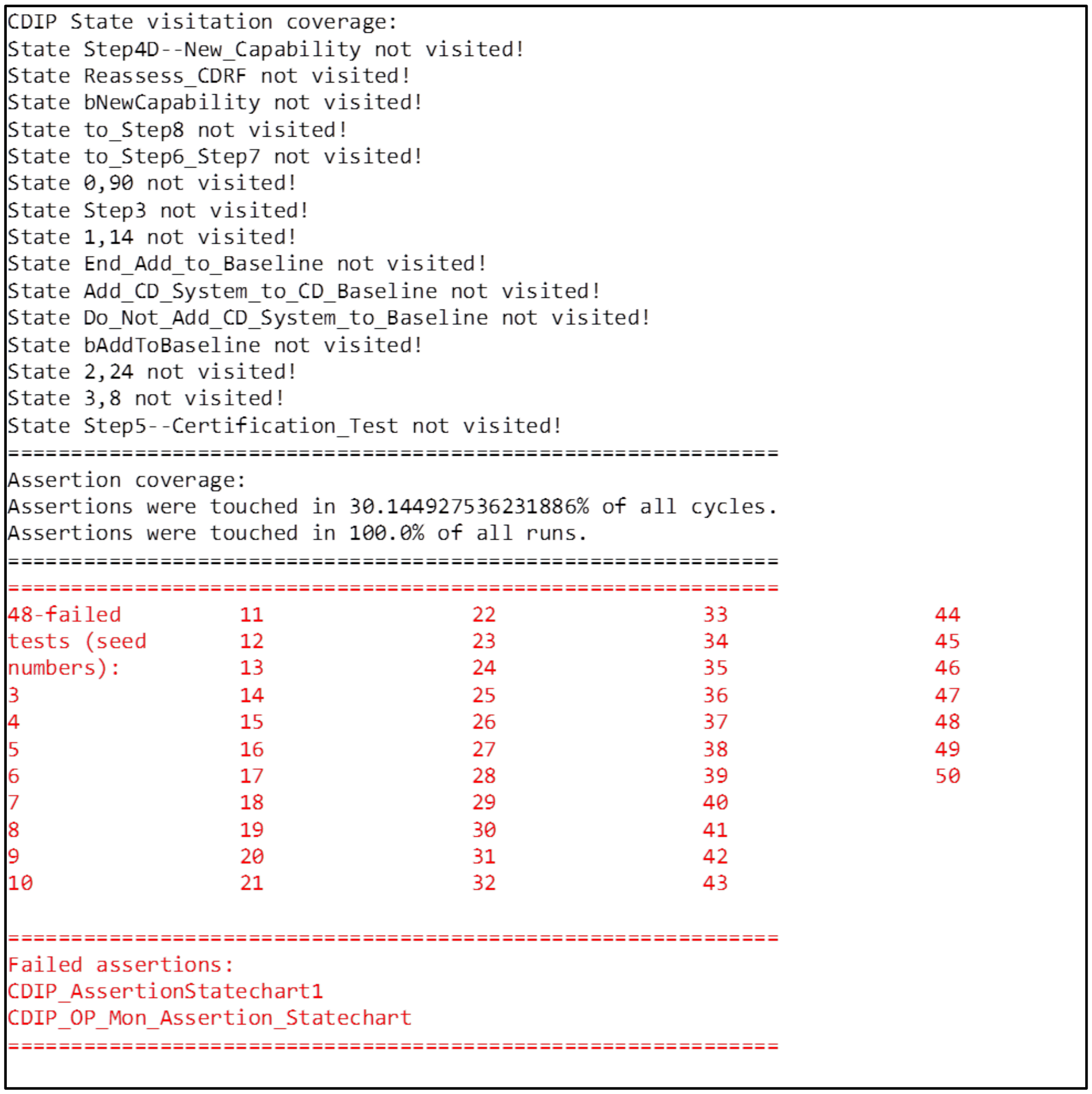
Runtime execution monitoring provides the process engineer with a means of exploring the effect of numerous input sequences on the process model during automated testing. The process engineer is able to adjust test parameters such as number of tests, test length, and number of permissible loops, in addition to using the information from testing to better understand, debug, maintain, and communicate with other engineers, the users, and the stakeholders about a process model.

Figure 45 and Figure 46 show a portion of the final results of two WTG tests runs on the CDIP process model. In this case, we set the number of tests to 50 and we are provided with feedback from the model (e.g., the output of println() statements embedded in the model), a listing of states within the model that were not entered during the test run, and a specific listing of which tests encountered one or more failed assertions (i.e., "47-failed tests" in Figure 45 and Figure 46). During automated testing with a large number of test runs, we expect most of the tests to have failed assertions since the WTG explores the possible paths for given set of inputs. Depending on what we are attempting to test in an automated test run, it may or may not be acceptable to have states not visited during test. For instance, if we wish to determine if the embedded statechart assertions fail and succeed for a given set of events and variables we may be able to accomplish this without visiting all states. If we wish to ensure that all states of the model are reachable from an input sequence then we would likely increase the number of test runs to ensure enough input sequences are presented to the executing model to fully visit all states.



```
CDIP State visitation coverage:
State Step4D--New_Capability not visited!
State Step10--Appellate_Process not visited!
State to_Step10 not visited!
State to_Step10 not visited!
State bInitAppeal not visited!
State 0,90 not visited!
State 1,14 not visited!
State Init not visited!
State Add_CD_System_to_CD_Baseline not visited!
State 2,24 not visited!
State 3,8 not visited!
State Step5--Certification_Test not visited!
============================================================
Assertion coverage:
Assertions were touched in 31.564245810055862% of all cycles.
Assertions were touched in 100.0% of all runs.
============================================================
============================================================
47-failed          11                 21                 31                 41
tests (seed        12                 22                 32                 42
numbers):          13                 23                 33                 43
3                  14                 24                 34                 44
5                  15                 25                 35                 45
6                  16                 26                 36                 46
7                  17                 27                 37                 47
8                  18                 28                 38                 48
9                  19                 29                 39                 49
10                 20                 30                 40                 50
============================================================
Failed assertions:
CDIP_AssertionStatechart1
CDIP_OP_Mon_Assertion_Statechart
============================================================
```

Figure 45. CDIP Test Results Example 1



```
CDIP State visitation coverage:
State Step4D--New_Capability not visited!
State Reassess_CDRF not visited!
State bNewCapability not visited!
State to_Step8 not visited!
State to_Step6_Step7 not visited!
State 0,90 not visited!
State Step3 not visited!
State 1,14 not visited!
State End_Add_to_Baseline not visited!
State Add_CD_System_to_CD_Baseline not visited!
State Do_Not_Add_CD_System_to_Baseline not visited!
State bAddToBaseline not visited!
State 2,24 not visited!
State 3,8 not visited!
State Step5--Certification_Test not visited!
=============================================================
Assertion coverage:
Assertions were touched in 30.144927536231886% of all cycles.
Assertions were touched in 100.0% of all runs.
=============================================================
=============================================================
48-failed          11                  22                  33                  44
tests (seed        12                  23                  34                  45
numbers):          13                  24                  35                  46
3                  14                  25                  36                  47
4                  15                  26                  37                  48
5                  16                  27                  38                  49
6                  17                  28                  39                  50
7                  18                  29                  40
8                  19                  30                  41
9                  20                  31                  42
10                 21                  32                  43

=============================================================
Failed assertions:
CDIP_AssertionStatechart1
CDIP_OP_Mon_Assertion_Statechart
=============================================================
```

Figure 46. CDIP Test Results Example 2

Figure 47 shows a portion of the statechart animation results produced during the test run of Figure 45. During model development, we use the textual and graphical feedback to debug the model and to ensure that the model behaves as expected under a wide variety of input scenarios.

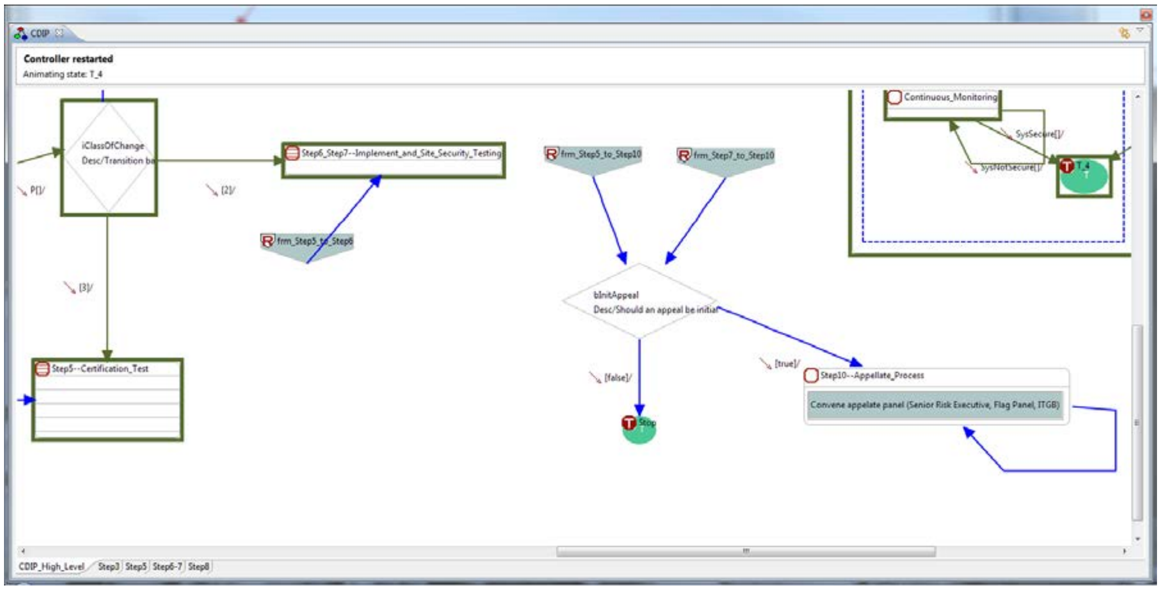


Figure 47. Graphic Feedback for CDIP Test Results Example 2

Visual test feedback helps the process engineer to better understand, develop, debug, and maintain the type of C&A processes we are interested in modeling. In addition, because visual representations are generally easier for humans to understand, they facilitate communication about the process under evaluation among users, stakeholders, and process engineers.

Table 6. Runtime Execution Monitoring Data Collection

Description of Attributes	***Test Run #***			
	1	**2**	**3**	**4**
Number of tests per run	5	25	50	100
Failed assertions	☑	☑	☑	☑
All states visited	☐	☐	☐	☐
Time to complete test run	15.9s	88.2s	144.2s	346.5s

As described in Section III.C.6.c, the data shown in Table 6 provided a means of comparing state visitation coverage and processing time across test runs. For this 100-test run the single flowchart box *Do_Not_Add_CD_System_to_Baseline* was the only state/flowchart box not visited. This prompted us to go back and review that portion of the model to ensure the results didn't indicate a problem with the model. In this case, the location of the *Do_Not_Add_CD_System_to_Baseline* flowchart box was such that the results made sense due to the location of the box relative to the flow of the model during automated testing.

B. CROSS DOMAIN SOLUTION WORKFLOW PROCESS

1. Process Selection

The CDS Workflow replaced the UCDMO's CDIP with the former representing a process-based initiative to federate the request, reuse, development, implementation, and C&A of cross domain solutions. The CDS Workflow is a more complex process than its predecessor with five major process blocks and four levels of hierarchy. As the successor to the CDIP, this is now the process by which designated authorities such as the Cross Domain Resolution Board (CDRB) decide whether to allow a given to CDS operate.

2. Process Analysis

In an effort to better document the CDS Workflow process, the UCDMO captured elements of it as UML use case (see Figure 48) and activity diagrams (see Figure 49) developed in Rational Rose Modeler[5]. The use case and activity diagrams were a starting point for analyzing and understanding the process, which facilitated the analysis phase of our modeling approach.

[5] Rational Rose is a commercial UML modeling tool developed by IBM. Additional information is available at http://www-01.ibm.com/software/awdtools/developer/rose/modeler/.

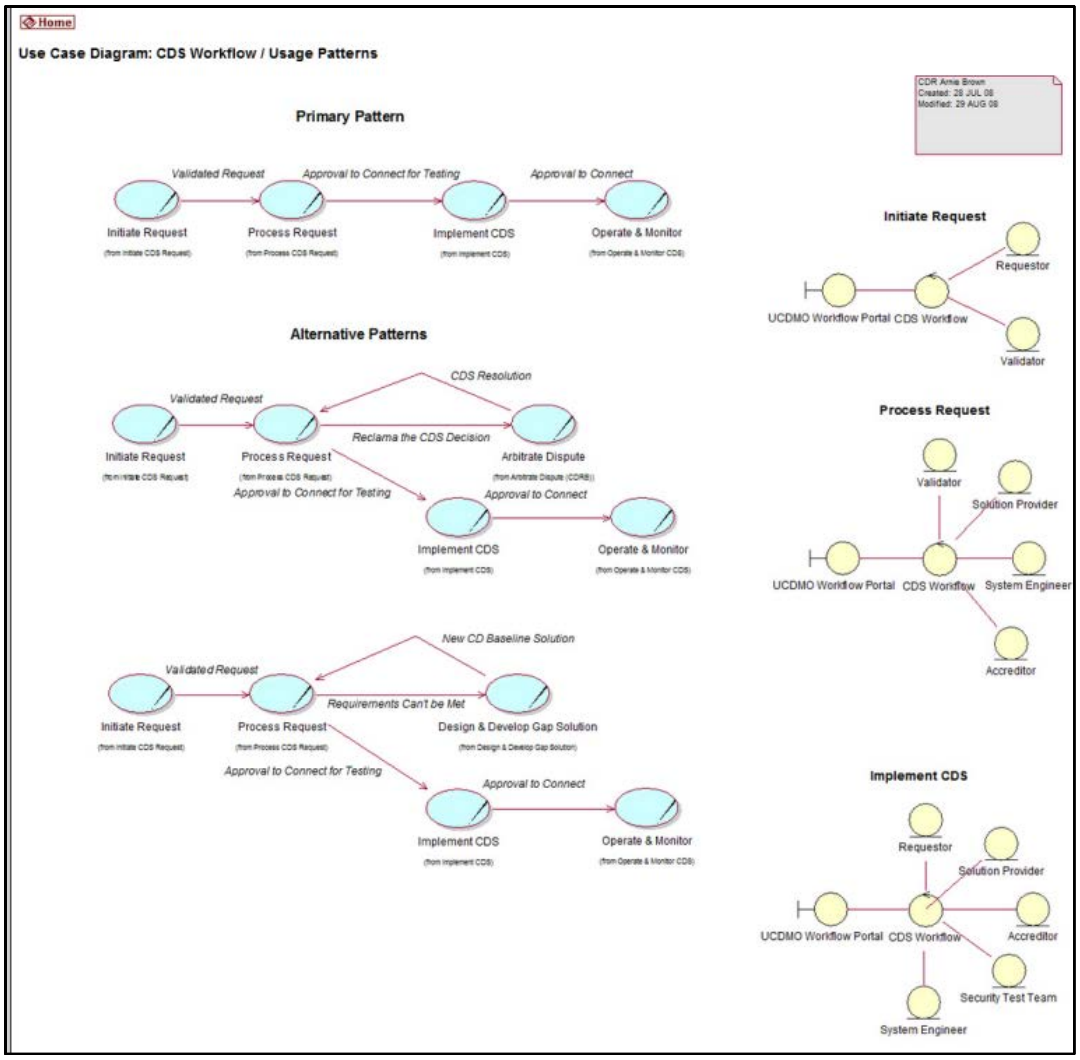


Figure 48. CDS Use Case Diagrams

The activity diagram shown in Figure 49 is for a sub-process of the larger CDS Workflow process. Activity diagrams are UML artifacts and the process of translating them is generally straightforward since activity diagrams can be directly translated to statecharts (Bruegge and Dutoit 2004, 62–67). Each activity diagram represents a sub-process within the overall CDS Workflow. Thus one of the challenges was to develop an understanding of how the different activity diagrams related to each other in terms of processes, sub-processes, and sequencing in order to provide us with the necessary information to build a model reflective of the actual CDS Workflow. Most of the CDS sub-processes had been documented in activity diagrams. However, the only documentation for the "Operate and Monitor" sub-process was a single point on the use case diagrams of Figure 48. This prompted us to hold further discussions with the UCDMO representatives in order to determine the flow, elements, and desired behaviors of this sub-process. This demonstrates one of the challenges of applying formal methods tools and technique to processes operating in real-world environments.

The use case diagrams of Figure 48 were particularly helpful for developing the proper sequencing of sub-processes within the overall CDS Workflow.

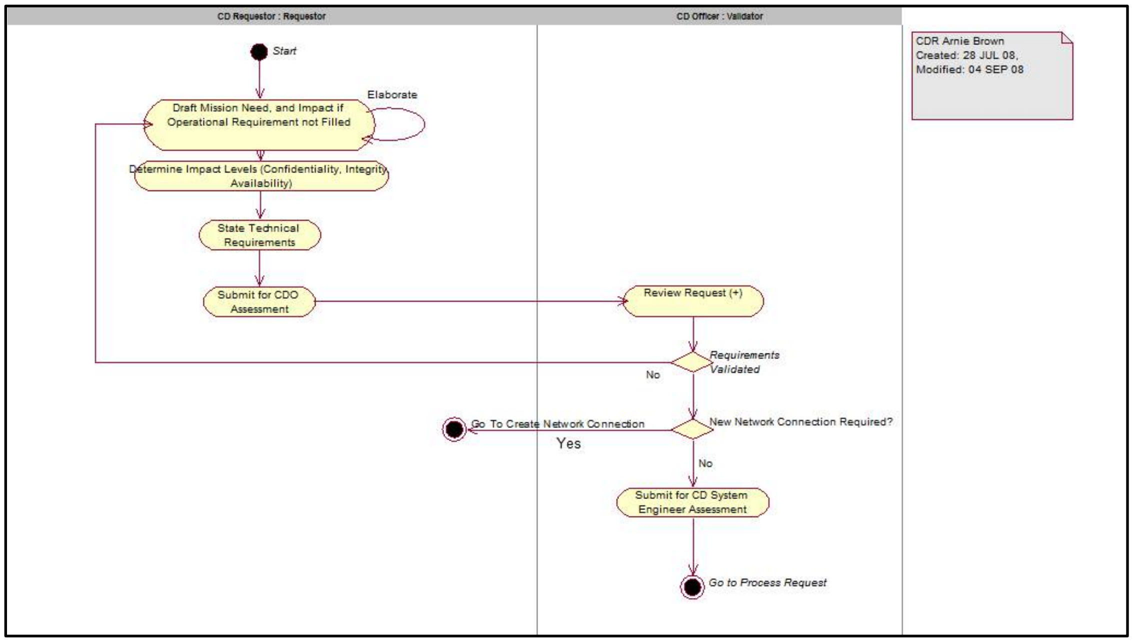

CD Requestor : Requestor
CD Officer : Validator
CDR Arnie Brown
Created: 28 JUL 08,
Modified: 04 SEP 08
Start
Elaborate
Draft Mission Need, and Impact if Operational Requirement not Filled
Determine Impact Levels (Confidentiality, Integrity, Availability)
State Technical Requirements
Submit for CDO Assessment
Review Request (+)
Requirements Validated
No
Go To Create Network Connection
Yes
New Network Connection Required?
No
Submit for CD System Engineer Assessment
Go to Process Request

Figure 49. Activity Diagram for Sub-Process Titled "Initiate Reqest"

As prescribed by our process modeling approach, we identified individual threads, decision points, and key elements for translation to the type of statechart artifacts used when building StateRover models (e.g., states, visual switches).

a. Threads

The CDS Workflow is a complex process with several sub-processes that have concurrent activities occurring. As discussed in Section III.A.a, we model concurrent activity using the statechart thread construct.

The vertical or horizontal parallel lines in an activity diagram denote swimlanes which provide a way to group activities performed by the same actor or to group activities in a single thread (Ambler 2005). We observed that in the UCDMO diagrams, groups of related activities have been grouped together within swimlanes, with some of these activities happening concurrently. We translated the swimlanes in each activity diagram to threads within appropriate states in the process model.

Using this approach, the activity diagram of Figure 50 shows two separate sets of activity. This will be modeled with two separate threads.

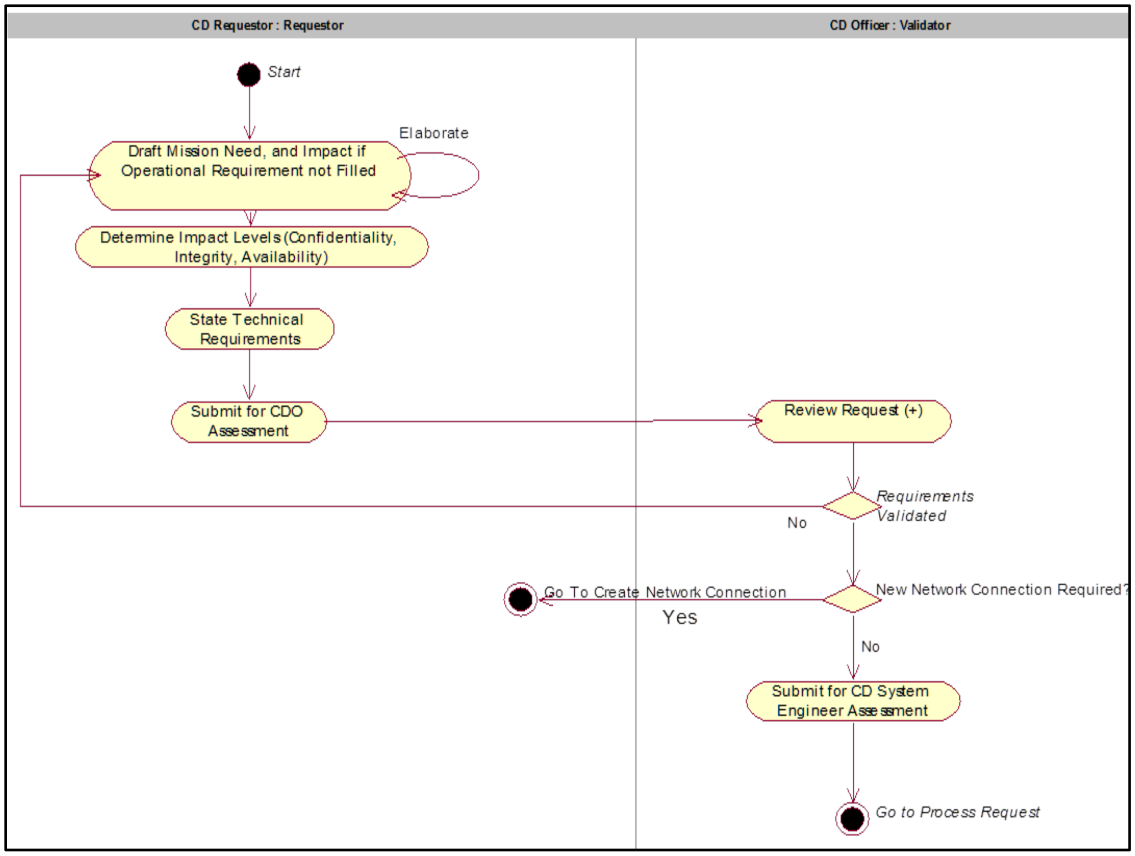

CD Requestor : Requestor
CD Officer : Validator
Start
Elaborate
Draft Mission Need, and Impact if Operational Requirement not Filled
Determine Impact Levels (Confidentiality, Integrity, Availability)
State Technical Requirements
Submit for CDO Assessment
Review Request (+)
Requirements Validated
No
Go To Create Network Connection
Yes
New Network Connection Required?
No
Submit for CD System Engineer Assessment
Go to Process Request

Figure 50. Initiate Request Activity Diagram

On the right hand side of Figure 50, *Review Request (+)* represents the sub-process shown in Figure 51. This activity diagram uses horizontal swimlanes to denote grouped activities; however, concurrent activities are taking place within each set of swimlanes. In this case, we model the activities within each set of swimlanes in a single state and use three threads and four threads for the top and bottom sets of swimlanes, respectively, to separate concurrent activity.

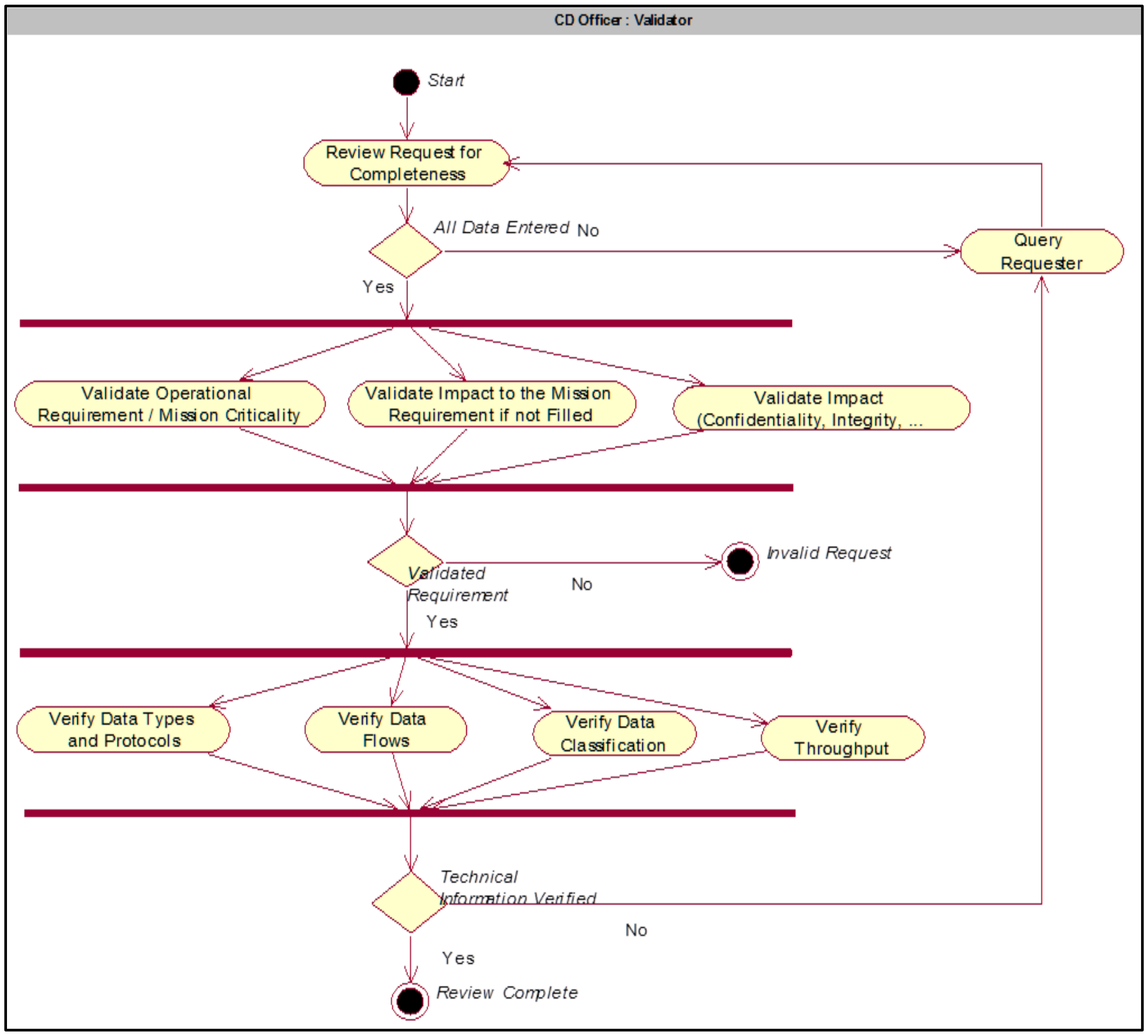

CD Officer : Validator
Start
Review Request for Completeness
All Data Entered
No
Query Requester
Yes
Validate Operational Requirement / Mission Criticality
Validate Impact to the Mission Requirement if not Filled
Validate Impact (Confidentiality, Integrity, ...
Validated Requirement
No
Invalid Request
Yes
Verify Data Types and Protocols
Verify Data Flows
Verify Data Classification
Verify Throughput
Technical Information Verified
No
Yes
Review Complete

Figure 51. Review Request Activity Diagram

The *Process CDS Request* activity diagram of Figure 52 shows that processing a CDS request involves four separate sets of activities. We model this with four separate threads.

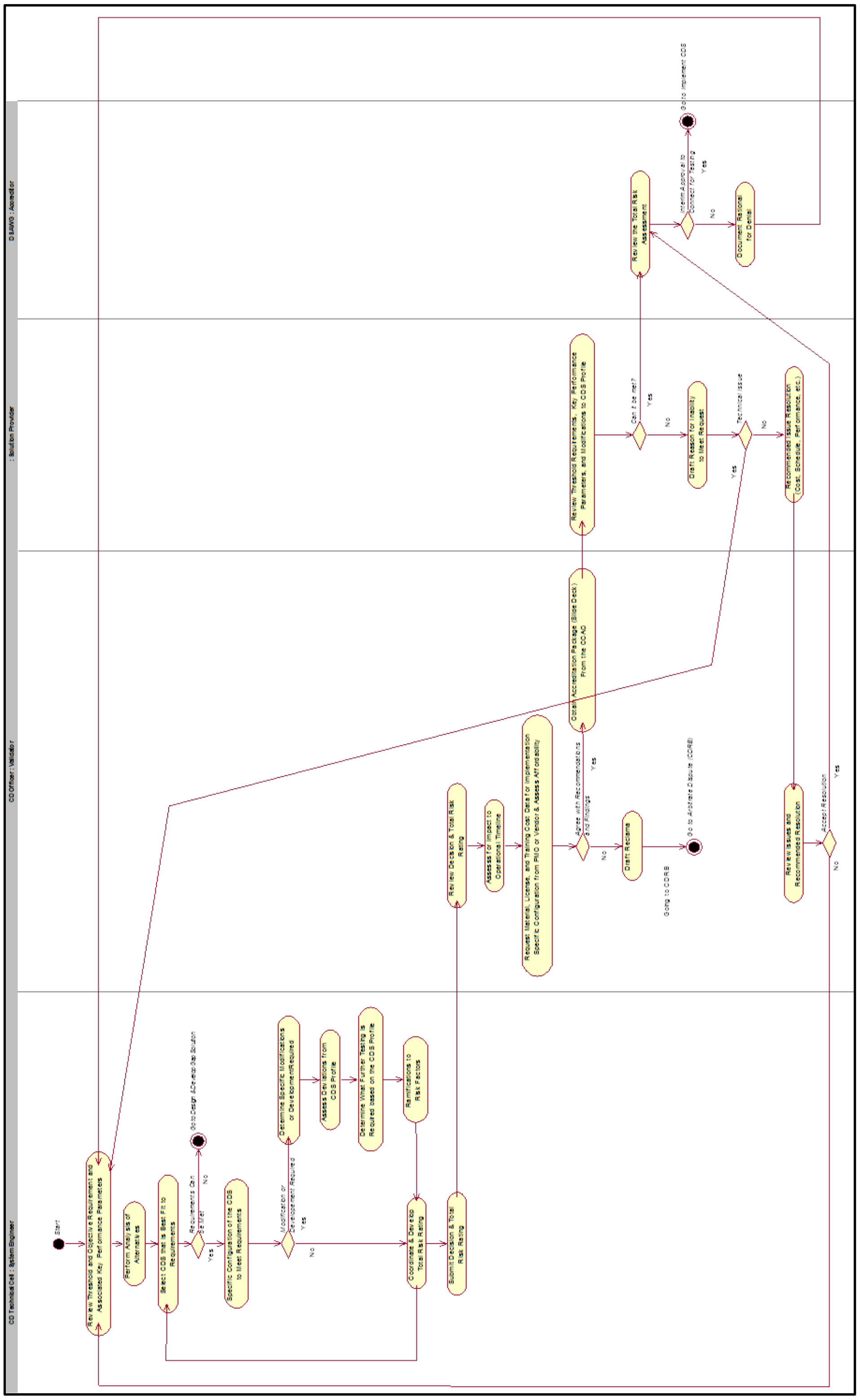
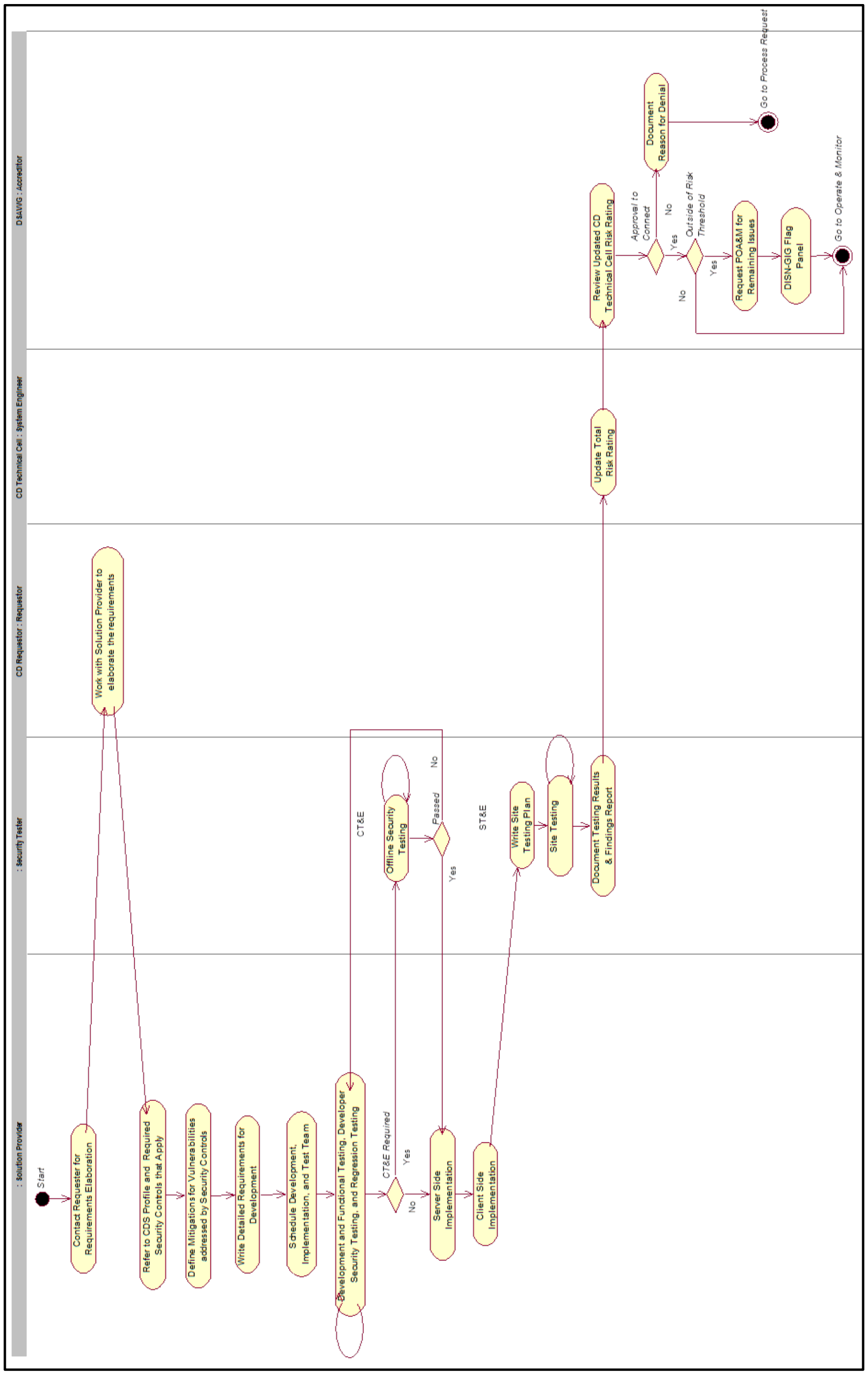

CD Technical Cell : System Engineer
CD Officer : Validator
: Solution Provider
DSAWG : Accreditor
Start
Review Threshold and Objective Requirement and Associated Key Performance Parameters
Perform Analysis of Alternatives
Select CDS that is Best Fit to Requirements
Requirements Can be Met?
Yes
No
Go to Design & Develop Solution
Specific Configuration of the CDS to Meet Requirements
Modification or Development Required
Yes
No
Determine Specific Modifications or Development Required
Assess Deviations from CDS Profile
Determine What Further Testing is Required based on the CDS Profile
Ramifications to Risk Factors
Coordinate & Develop Total Risk Rating
Submit Decision & Total Risk Rating
Review Decision & Total Risk Rating
Assess for Impact to Operational Timeline
Request Material, License, and Training Cost Data for Implementation Specific Configuration from PMO or Vendor & Assess Affordability
Agree with Recommendations and Findings
Yes
No
Draft Reclama
Go to Arbitrate Dispute (CDRB)
Going to CDRB
Obtain Accreditation Package (Slide Deck) From the CDAO
Review Threshold Requirements, Key Performance Parameters, and Modifications to CDS Profile
Can it be met?
Yes
No
Draft Reason for Inability to Meet Request
Technical Issue
Yes
No
Recommended Issue Resolution (Cost, Schedule, Performance, etc.)
Review Issues and Recommended Resolution
Accept Resolution
Yes
No
Review the Total Risk Assessment
Interim Approval to Connect for Testing
Yes
No
Document Rational for Denial
Go to Implement CDS

Figure 52. Process CDS Request Activity Diagram



: Solution Provider
: Security Tester
CD Requestor : Requestor
CD Technical Cell : System Engineer
DSAWG : Accreditor
Start
Contact Requester for Requirements Elaboration
Work with Solution Provider to elaborate the requirements
Refer to CDS Profile and Required Security Controls that Apply
Define Mitigations for Vulnerabilities addressed by Security Controls
Write Detailed Requirements for Development
Schedule Development, Implementation, and Test Team
Development and Functional Testing, Developer Security Testing, and Regression Testing
CT&E Required
Yes
No
CT&E
Offline Security Testing
Passed
Yes
No
Server Side Implementation
Client Side Implementation
ST&E
Write Site Testing Plan
Site Testing
Document Testing Results & Findings Report
Update Total Risk Rating
Review Updated CD Technical Cell Risk Rating
Approval to Connect
No
Yes
Document Reason for Denial
Go to Process Request
Outside of Risk Threshold
Yes
No
Request POA&M for Remaining Issues
DISN-GIG Flag Panel
Go to Operate & Monitor

Figure 53. Implement CDS Request Activity Diagram

Implementing a CDS has five separate sets of activity occurring, as shown in Figure 53. We model this with five separate threads.

b. Decision Points

Next, we examine the CDS Workflow process to determine decision points and use Table 7 to manage and track them.

Table 7. CDS Workflow Decision Points

Description	Variable Name	Possible Values	In Model
Decide whether new network connection is required to satisfy CDS request.	bNewNetConnectRqrd	True, false	☒
Did CDS requestor enter all necessary data on CDS request?	bAllDataEntered	True, false	☒
Is CDS requirement valid (CD Officer Validator)?	bValidatedRqrmt	True, false	☒
Has all technical information on CDS request been verified?	bTechInfoVerified		☒
Decide whether the “best-fit” CDS meets the requested requirement.	bCanMeetReq	True, false	☒
Does the “best-fit” CDS require modification to meet requirements?	bisModRqrd	True, false	☒
Does CDS decision-making body agree with recommendations and findings?	bAgreeRecommFind	True, false	☒
Does CD Officer Validator accept recommended issue resolution?	bAcceptRes	True, false	☒
Can threshold requirements be met with proposed configuration?	bThrshldRqrmtsMet	True, false	☒
Do technical issues prevent meeting the requirement?	bTechIssues	True, false	☒
After reviewing risk assessment, does CDS have interim approval to connect for testing?	bIntApprovConnctTstng	True, false	☒
Are certification, testing, and evaluation required?	bCTE_Req	True, false	☒

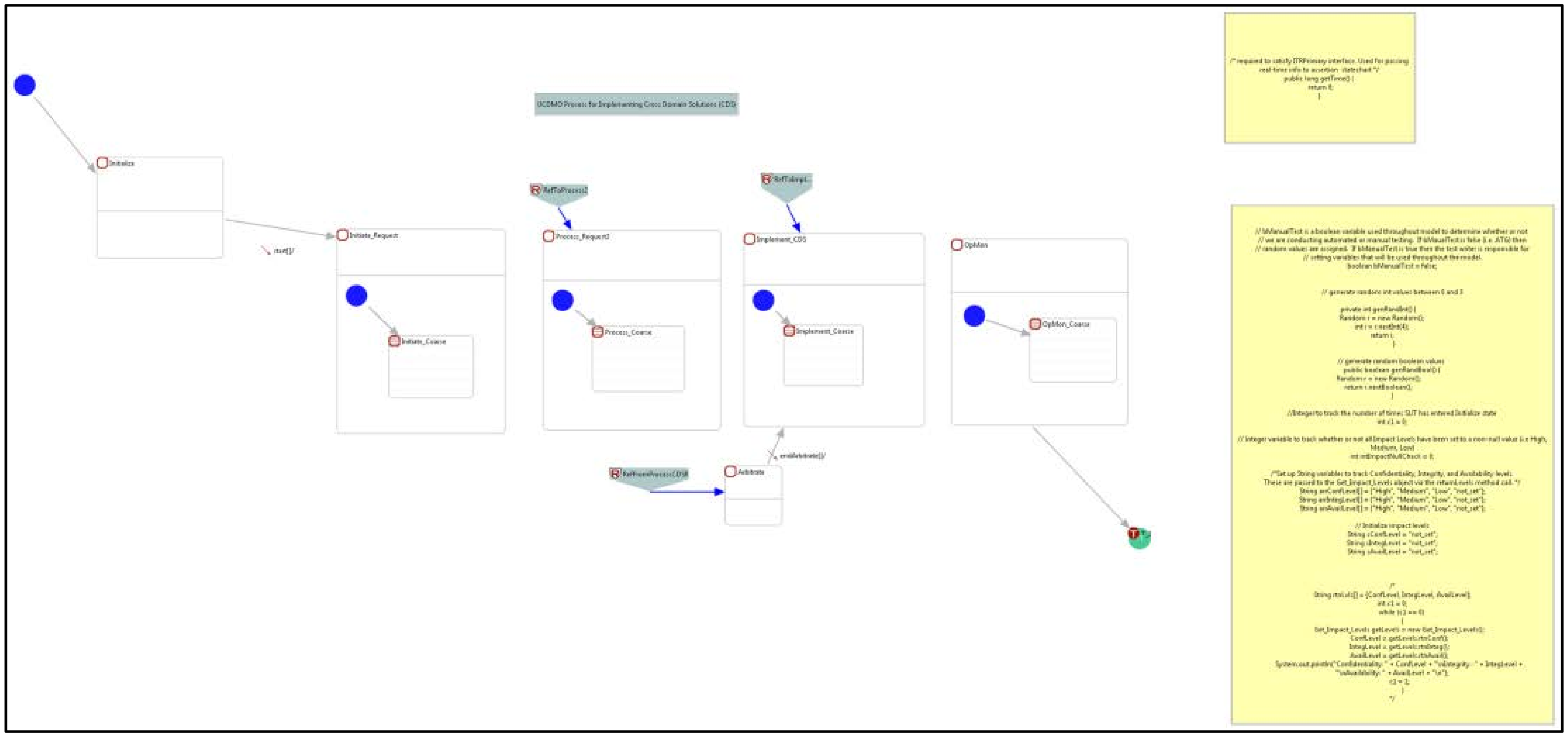
Description	**Variable Name**	**Possible Values**	**In Model**
Did CDS pass offline testing?	bPassOfflineTest	True, false	☒
Is approval granted to connect implemented CDS to network(s)?	bApprovetoConnect	True, false	☒
Is CDS outside of risk threshold?	bOutsideRiskThresh	True, false	☒

c. Layers

The CDS Workflow is composed of processes and sub-processes. Upon analyzing the process we determined that it would be best to represent the process using four levels of hierarchy. Each of the "Initiate Request," "Process Request," "Implement CDS," and "Operate and Monitor" sub-processes were of sufficient complexity to warrant individual articulation as a nested process at a second level of hierarchy. In addition, the "Review Request" sub-process of "Initiate Request" was sufficiently complex for individual articulation at a third level of hierarchy.

3. Construct Process Model

In this section, we bring together the results of process selection an analysis to articulate the formal process model of the CDS Workflow. The diagram of Figure 54 shows the top-level view of the final process model.



```
/* required to satisfy ITRPrimary interface. Used for passing
real-time info to assertion statechart */
public long getTime() {
return 0;
}
```

```
// bManualTest is a boolean variable used throughout model to determine whether or not
// we are conducting automated or manual testing. If bManualTest is false (i.e. ATG) then
// random values are assigned. If bManualTest is true then the test writer is responsible for
// setting variables that will be used throughout the model.
boolean bManualTest = false;

// generate random int values between 0 and 3

private int genRandInt() {
Random r = new Random();
int i = r.nextInt(4);
return i;
}

// generate random boolean values
public boolean genRandBool() {
Random r = new Random();
return r.nextBoolean();
}

//Integer to track the number of times SUT has entered Initialize state
int c1 = 0;

// Integer variable to track whether or not all Impact Levels have been set to a non-null value (i.e. High,
Medium, Low)
int intImpactNullCheck = 0;

/*Set up String variables to track Confidentiality, Integrity, and Availability levels.
These are passed to the Get_Impact_Levels object via the returnLevels method call. */
String arrConfLevel[] = {"High", "Medium", "Low", "not_set"};
String arrIntegLevel[] = {"High", "Medium", "Low", "not_set"};
String arrAvailLevel[] = {"High", "Medium", "Low", "not_set"};

// Initialize impact levels
String sConfLevel = "not_set";
String sIntegLevel = "not_set";
String sAvailLevel = "not_set";


/*
String rtnLvls[] = {ConfLevel, IntegLevel, AvailLevel};
int c1 = 0;
while (c1 == 0)
{
Get_Impact_Levels getLevels = new Get_Impact_Levels();
ConfLevel = getLevels.rtnConf();
IntegLevel = getLevels.rtnInteg();
AvailLevel = getLevels.rtnAvail();
System.out.println("Confidentiality: " + ConfLevel + "\nIntegrity: " + IntegLevel +
"\nAvailability: " + AvailLevel + "\n");
c1 = 1;
}
*/
```

Figure 54. Top-level Statechart Model of CDS Workflow Process

Throughout model development and with the addition of each new step we perform two actions designed to ensure the model adheres to the underlying rules for semantic correctness: (i) Use the “Diagram/Validate” menu option to initiate StateRover’s validation routine and reveal detected errors (ii) initiate the code generation process. As detailed in Chapter III.C, the diagram validation and code generation provide an end-to-end semantic check of the model and identify errors.

For this process, our analysis indicated that the top-level of the model would be relatively simple in terms of the number of states and transitions. We initially constructed the model shown in Figure 55. This portion of the model ended up being very close to the final top-level view.

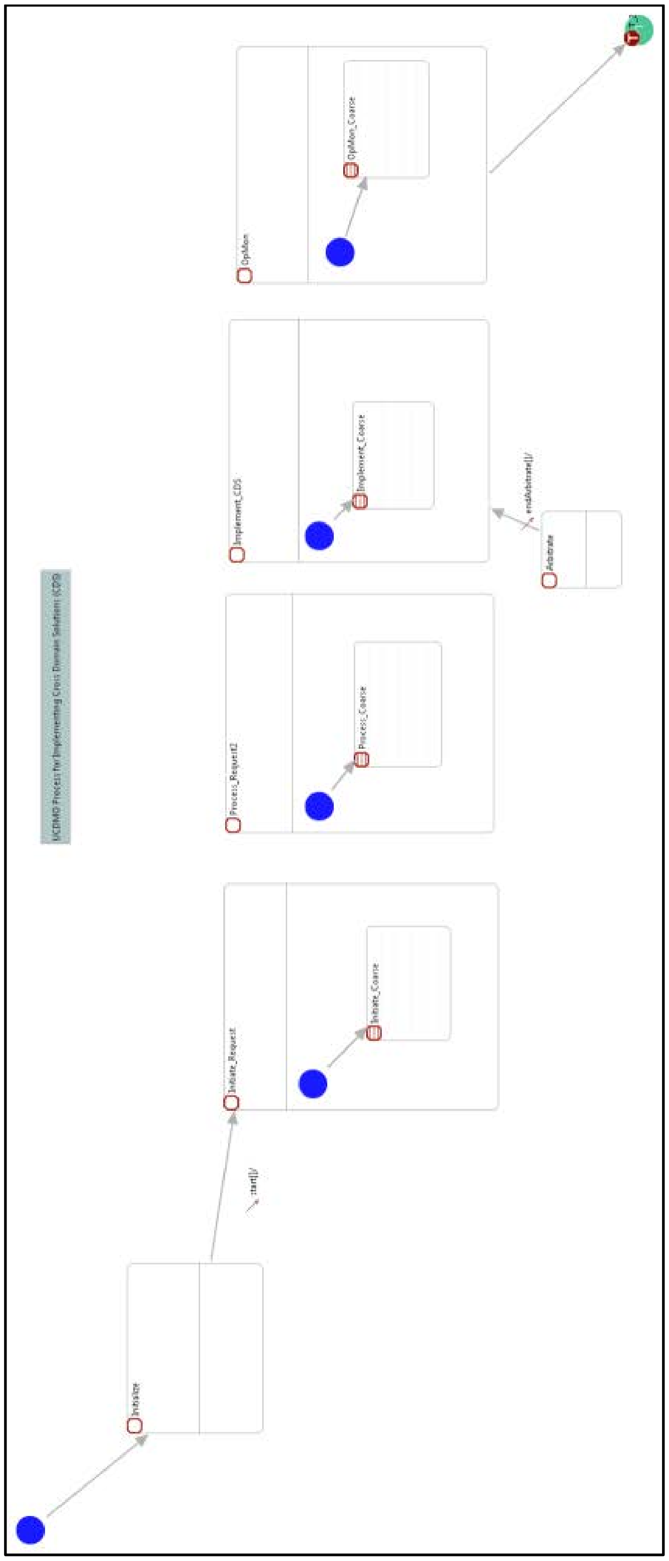


Figure 55. CDS Workflow Top-level View During Model Development

Next, we constructed in turn each of the detail views for the *Initiate_Coarse, Process_Coarse, Implement_Coarse,* and *OpMon_Coarse* states of Figure 55.

The detail view of *Initiate_Coarse* is shown in Figure 56. The middle thread is a placeholder for an embedded statechart assertion to be placed later in the modeling process. Just as occurred when constructing the CDIP process model, we run the validation and code generation routines at each step of model building to iteratively ensure the model is semantically and syntactically correct. We do this at each stage of model development.

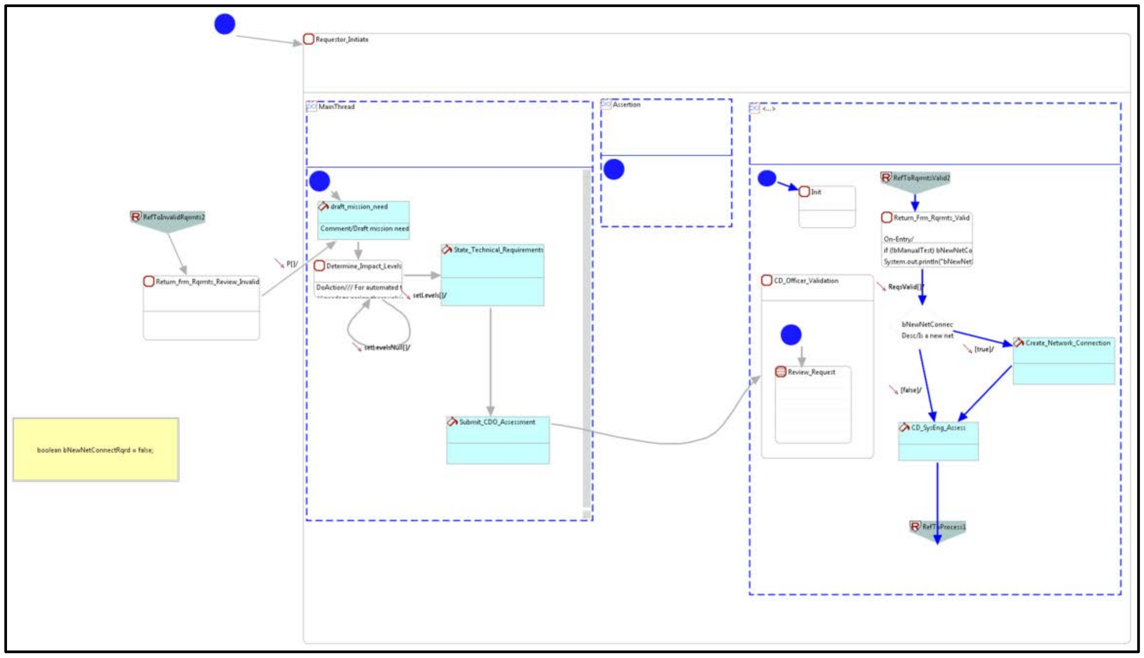


Figure 56. Initiate_Coarse

The right hand thread contains a coarse state, *Review_Request*, the details of which are shown in Figure 57. This sub-process is fully contained within the *Initiate_Coarse* state and therefore decomposes to the third level of hierarchy. We discussed complexity and layering in Chapter III of this document. Modeling of the second and third levels of hierarchy in this manner is an example of how we are able to drill down to successively finer levels of detail within the process model in order to deal with complexity.

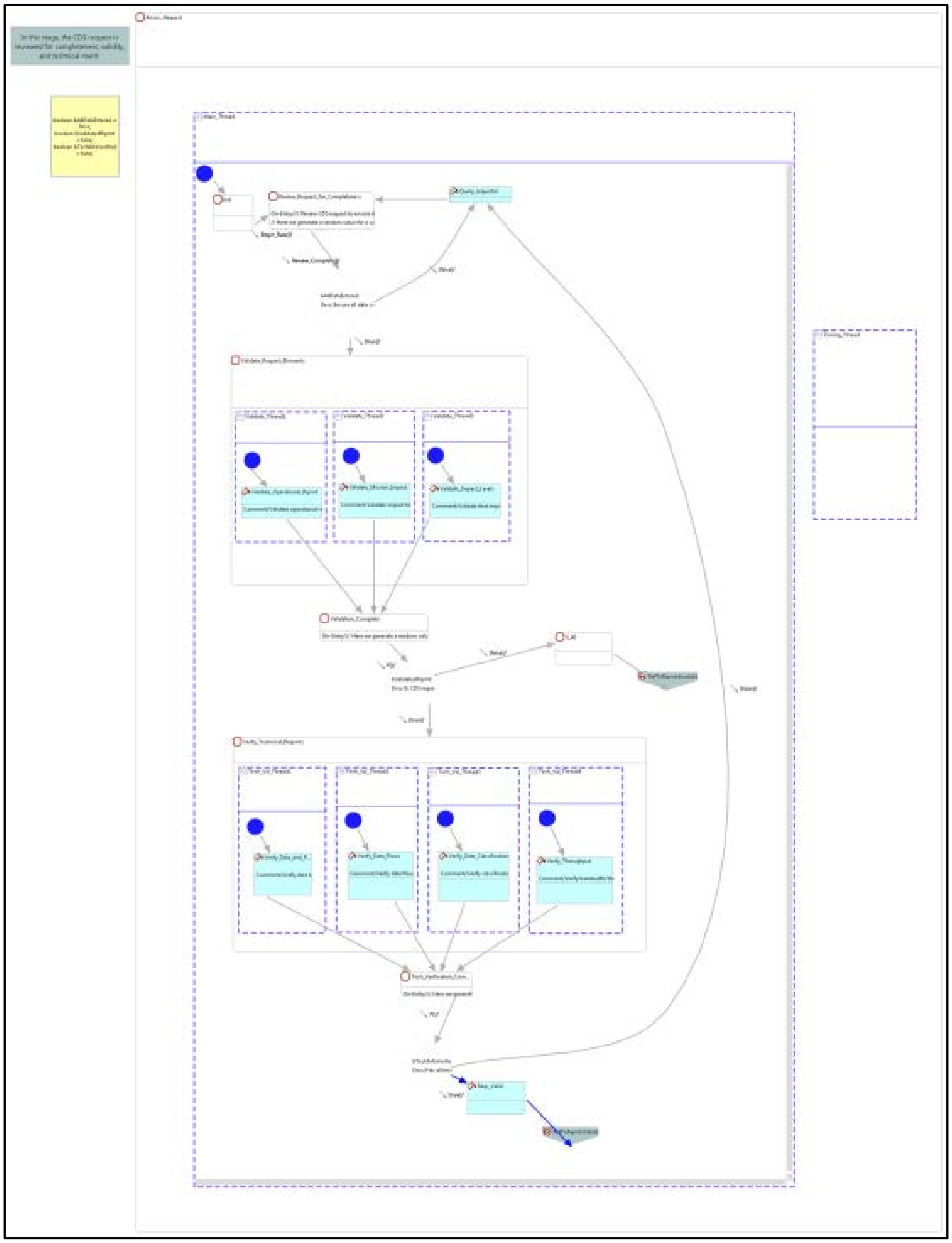

In this stage, the CDS request is reviewed for completeness, validity, and technical merit.
Main_Thread
Review_Request_for_Completeness
Begin_Test()
Review_Complete()
Validate_Request_Elements
Validation_Complete
Verify_Technical_Request
Timing_Thread

Figure 57. Review_Request

Next we constructed the detailed view of state *Process_Coarse*. This state has four separate threads, seven decision points, a mix of workflow and statechart elements, and a number of transitions between the threads. It was the most complex sub-process that we had modeled. Running the validation and code generation routines was particularly helpful to ensure the semantic and syntactic correctness as we iteratively built it. The detailed view of *Process_Coarse* is shown in Figure 58.

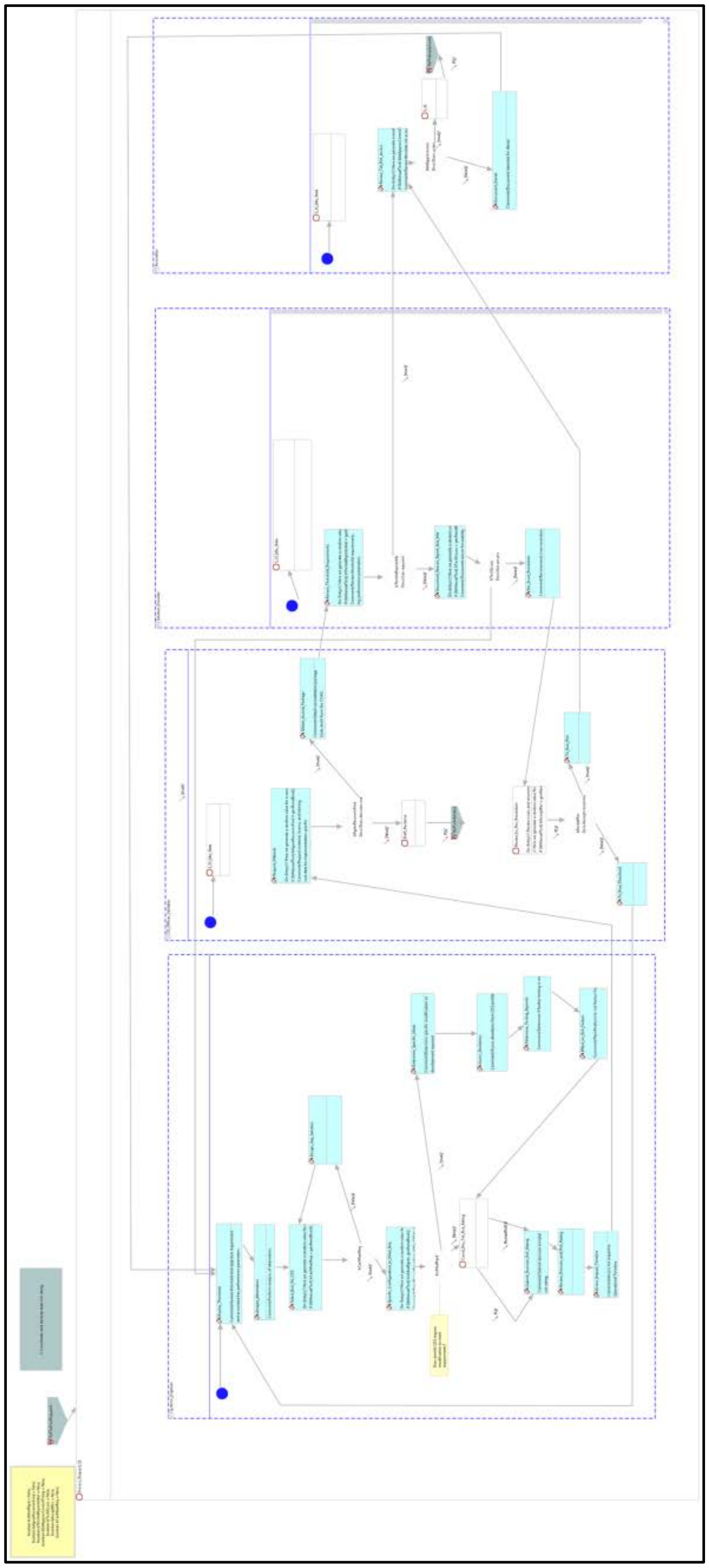


Figure 58. Process_Coarse

The next state modeled was *Implement_Coarse*. This sub-process has five threads and four decision points. As with the previous sub-process, our modeling approach of iterative validation and code generation helped us rapidly construct this portion of the model with no syntactic or semantic errors. This sub-process is shown in Figure 59.

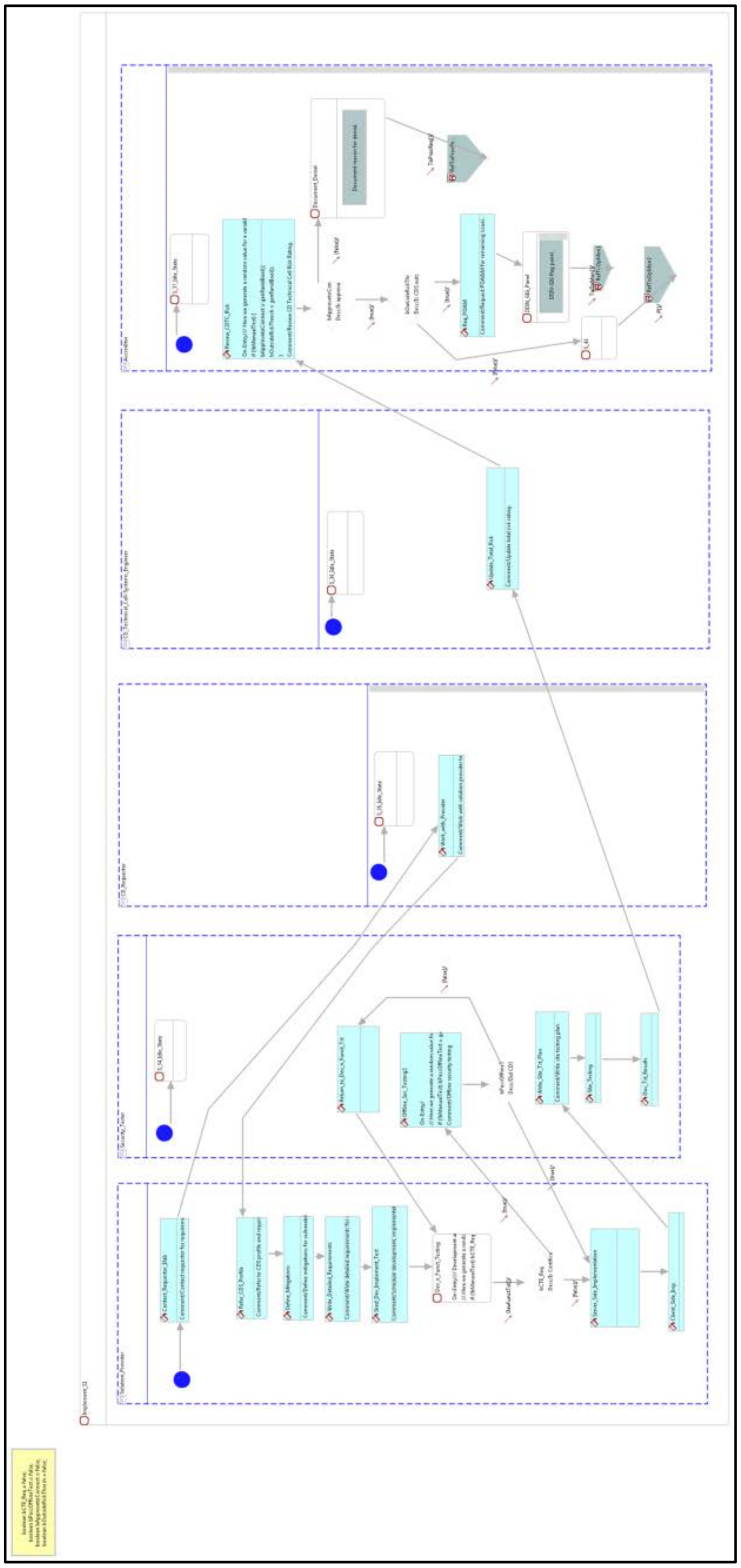


Figure 59. Implement_Coarse

The final detail view developed was for *OpMon_Coarse* sub-process. The detailed view of *OpMon_Coarse* is shown in Figure 60.

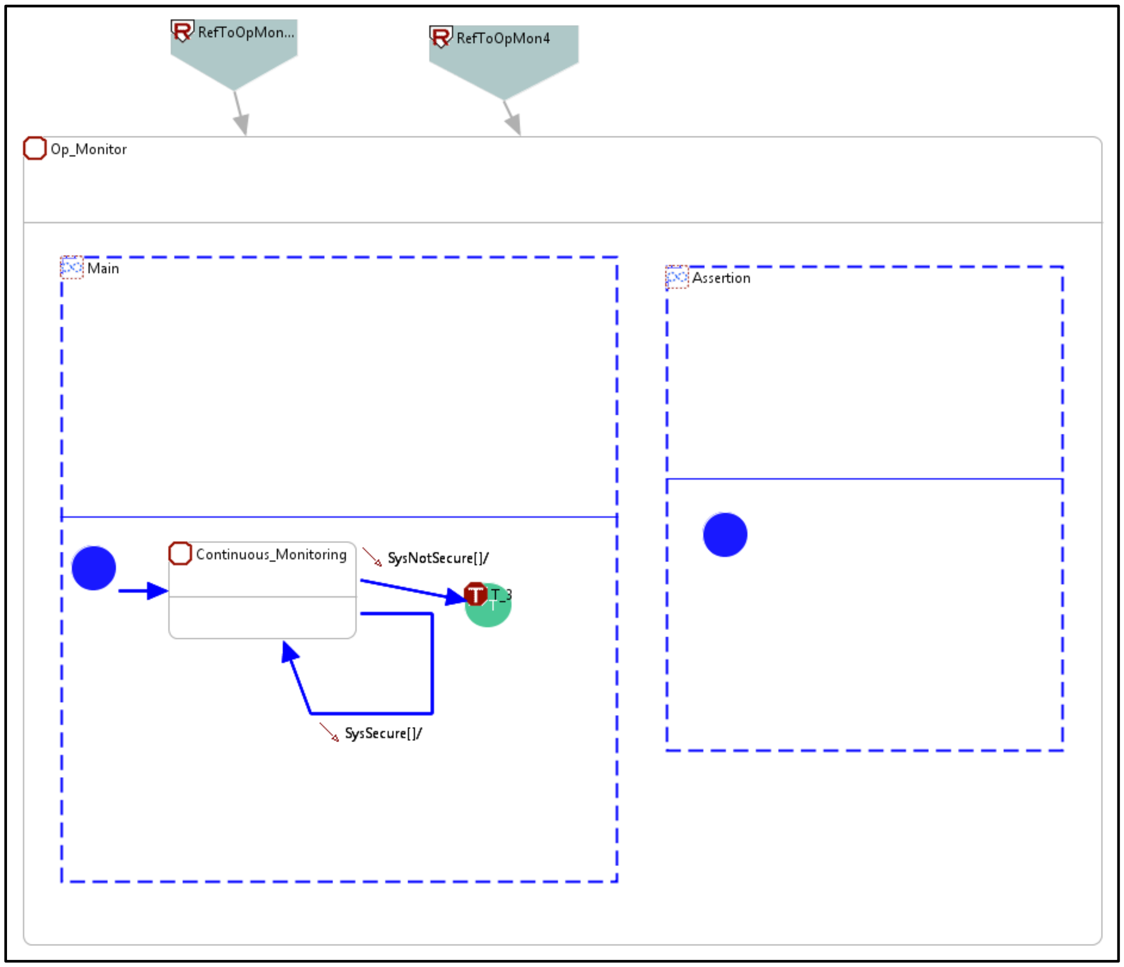


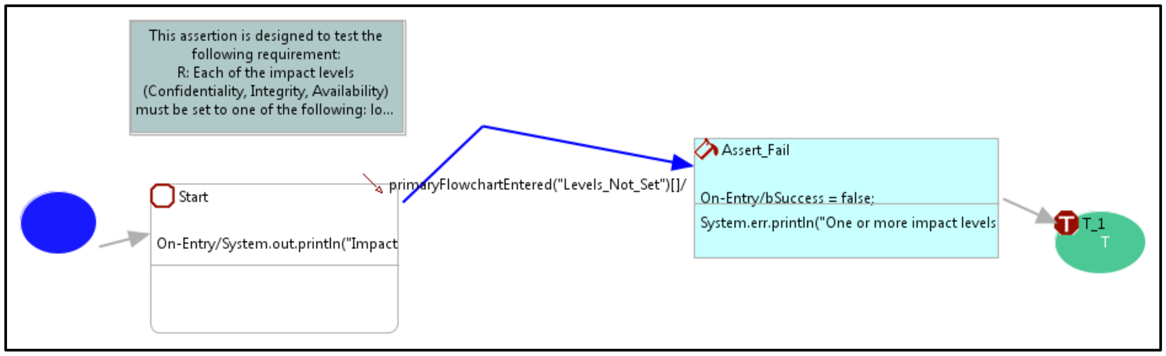
Figure 60. OpMon_Coarse

We intend to embed a statechart assertion in *OpMon_Coarse* later in the modeling process so we have positioned a placeholder thread to be filled in during the "Embed Assertions in Process Model" phase of our modeling approach.

4. Construct Statechart assertions

In this section we show three embedded assertions statecharts, each designed to model and enforce a different natural language requirement. This demonstrates the technical feasibility of applying statechart embedded assertions to the type of partially automated human-based C&A processes modeled with our approach.

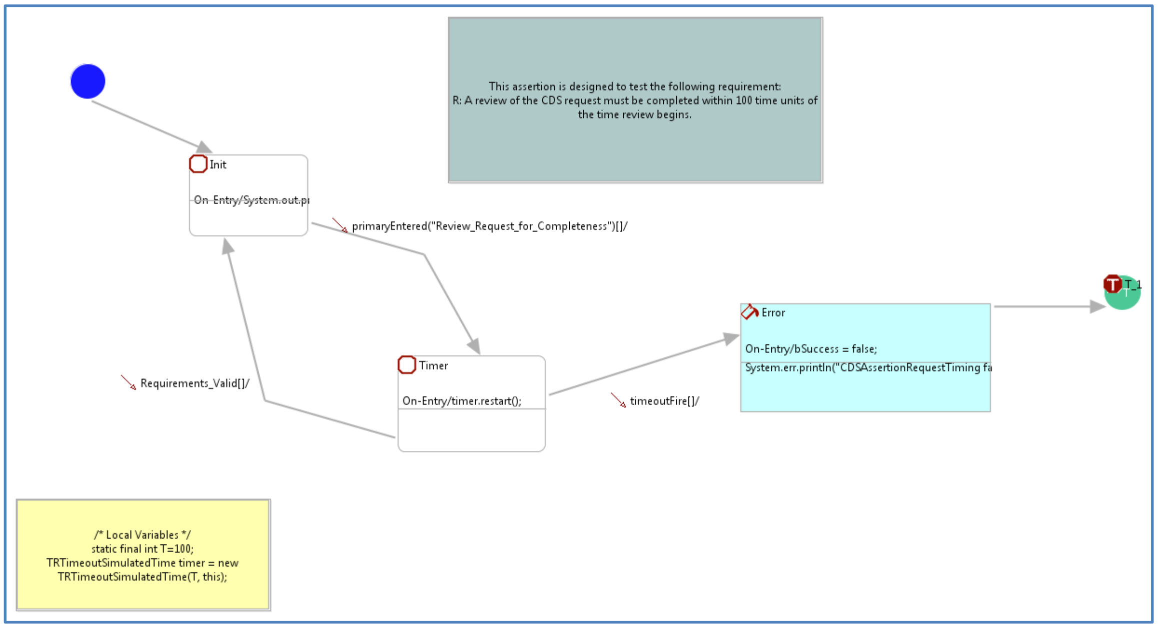
The statechart assertion of Figure 61 is a formal specification of the natural language requirement R1: *Each of the impact levels (Confidentiality, Integrity, Availability) must be set to one of the following: low, moderate, high*.


This assertion is designed to test the following requirement:
R: Each of the impact levels (Confidentiality, Integrity, Availability) must be set to one of the following: lo...

```
On-Entry/System.out.println("Impact
On-Entry/bSuccess = false;
System.err.println("One or more impact levels
```

Figure 61. Statechart Assertion for Requirement R1

The statechart assertion of Figure 62 is a formal specification of the natural language requirement R2: *Review of the CDS request must be completed within 100 time units of the time review begins*. This statechart assertion demonstrates the ability to apply enforceable temporal constraints to our model.


This assertion is designed to test the following requirement:
R: A review of the CDS request must be completed within 100 time units of the time review begins.

```
On-Entry/System.out.pı
On-Entry/timer.restart();
On-Entry/bSuccess = false;
System.err.println("CDSAssertionRequestTiming fa
```

```
/* Local Variables */
static final int T=100;
TRTimeoutSimulatedTime timer = new
TRTimeoutSimulatedTime(T, this);
```

Figure 62. Statechart Assertion for Requirement R2

The statechart assertion of Figure 63 is a formal specification of the natural language requirement R3: *The CDS implementation must remain secure during the “operate and monitor” phase of its lifecycle*.

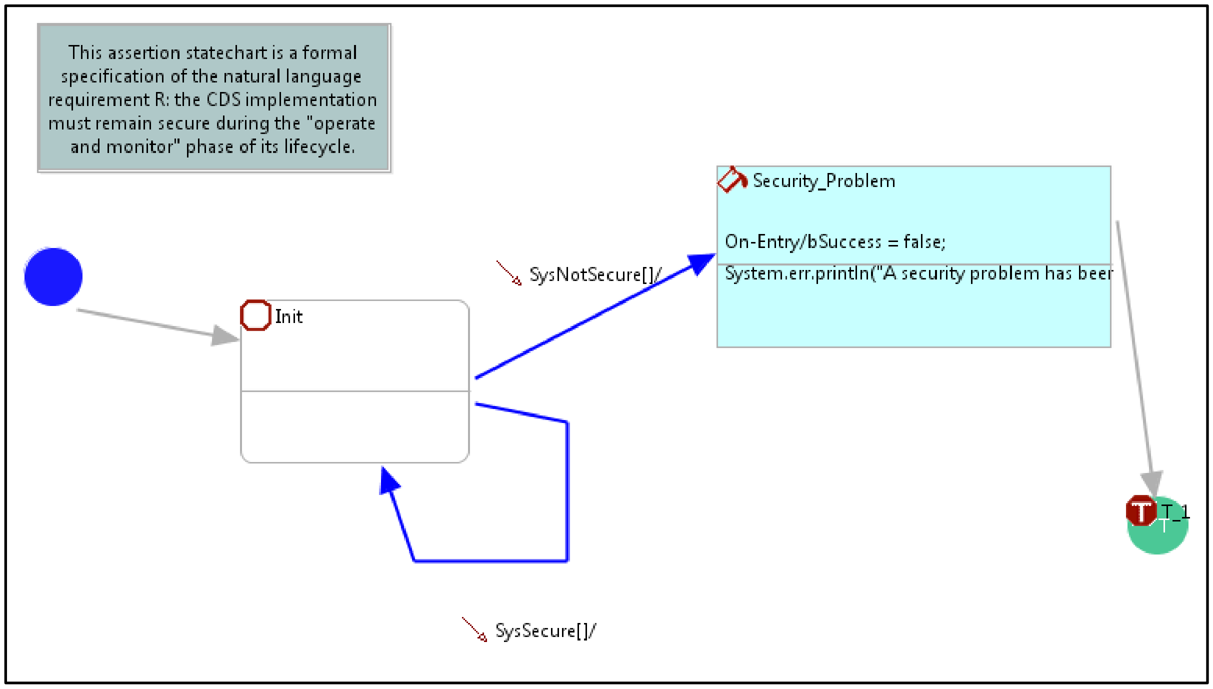

This assertion statechart is a formal specification of the natural language requirement R: the CDS implementation must remain secure during the "operate and monitor" phase of its lifecycle.
Security_Problem
On-Entry/bSuccess = false;
System.err.println("A security problem has beer
SysNotSecure[]/
Init
T_1
SysSecure[]/

Figure 63. Statechart Assertion for Requirement R3

5. Embed Assertions in Process Model

The next step in our process modeling approach is to embed the statechart assertions into the process model. As discussed in Section III.C.5 of this document, the process engineer places embedded statechart assertions within the model based on the desired scope of the assertion.

The statechart assertion of Figure 64 is only applicable to activity modeled in state *Requestor_Initiate*; therefore, to keep it appropriately scoped it is embedded in a thread within this state via the sub-statechart artifact of our chosen modeling tool as shown in Figure 65.

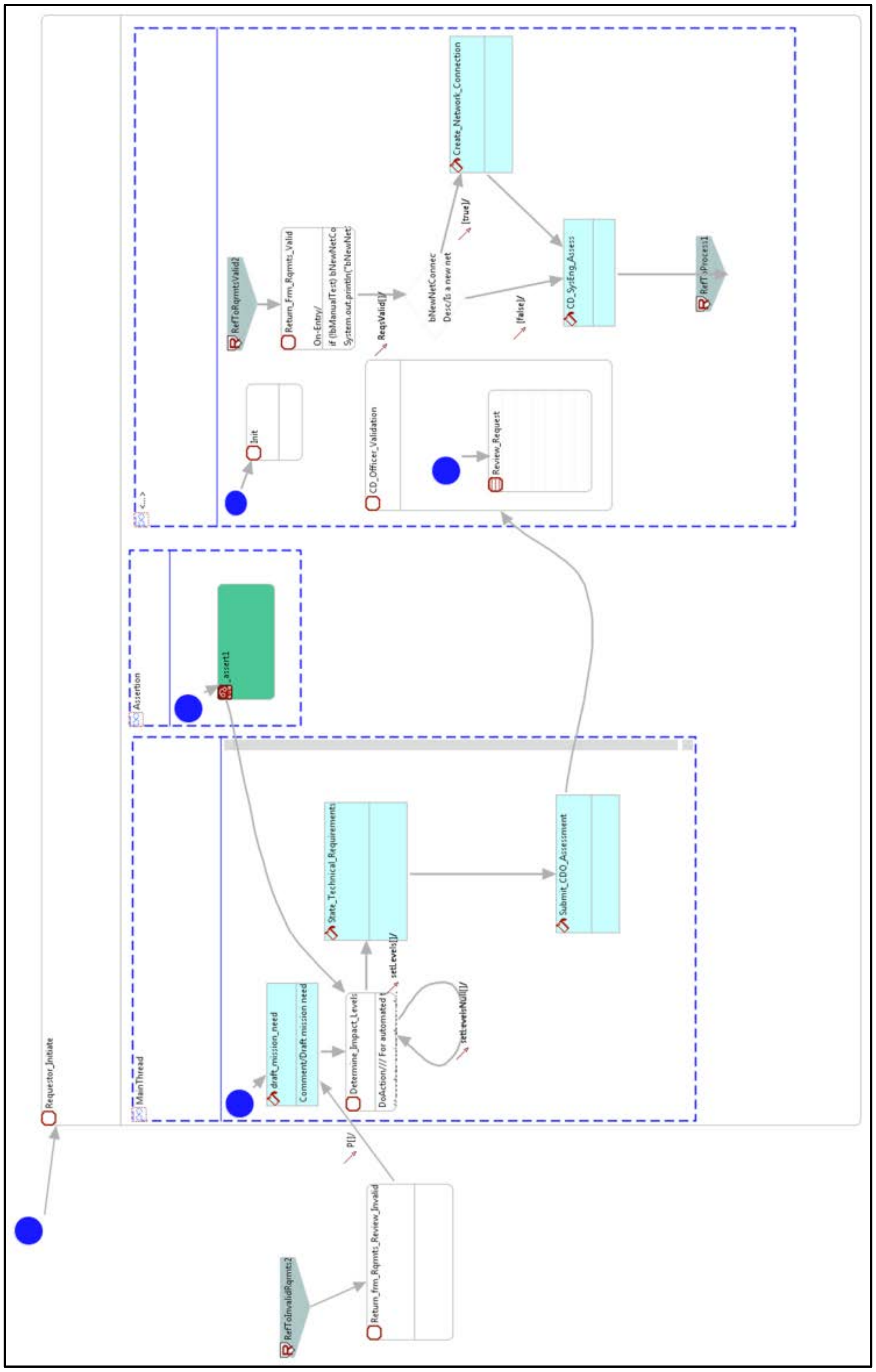


Figure 64. State *Requestor_Initiate* with Embedded Assertion

The statechart assertion for R2, shown in Figure 62, is only applicable to the activities and events occurring in state *Assess_Request* so it has been embedded within a thread (see Figure 65) of this state to maintain the appropriate scope.

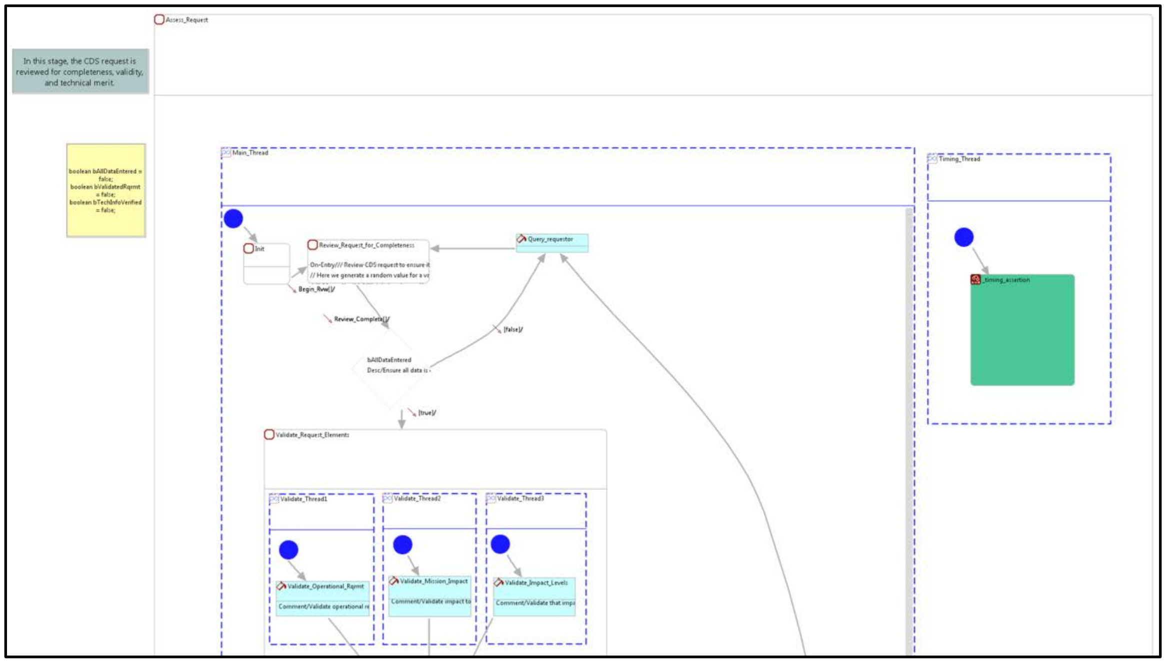

Assess_Request
In this stage, the CDS request is reviewed for completeness, validity, and technical merit.
boolean bAllDataEntered = false;
boolean bValidatedRqmt = false;
boolean bTechInfoVerified = false;
Main_Thread
Init
Review_Request_for_Completeness
On-Entry/// Review CDS request to ensure it
// Here we generate a random value for a va
Query_requestor
Begin_Rvw()/
Review_Complete()/
[false]/
bAllDataEntered
Desc/Ensure all data is
[true]/
Validate_Request_Elements
Validate_Thread1
Validate_Thread2
Validate_Thread3
Validate_Operational_Rqmt
Comment/Validate operational re
Validate_Mission_Impact
Comment/Validate impact to
Validate_Impact_Levels
Comment/Validate that impa
Timing_Thread
_timing_assertion

Figure 65. State *Assess_Request* with Embedded Statechart Assertion

The statechart assertion for R3, shown in Figure 63, is only applicable to the activities and events occurring in state *Op_Monitor* so it has been embedded within a thread (see Figure 66) of this state to maintain the appropriate scope.

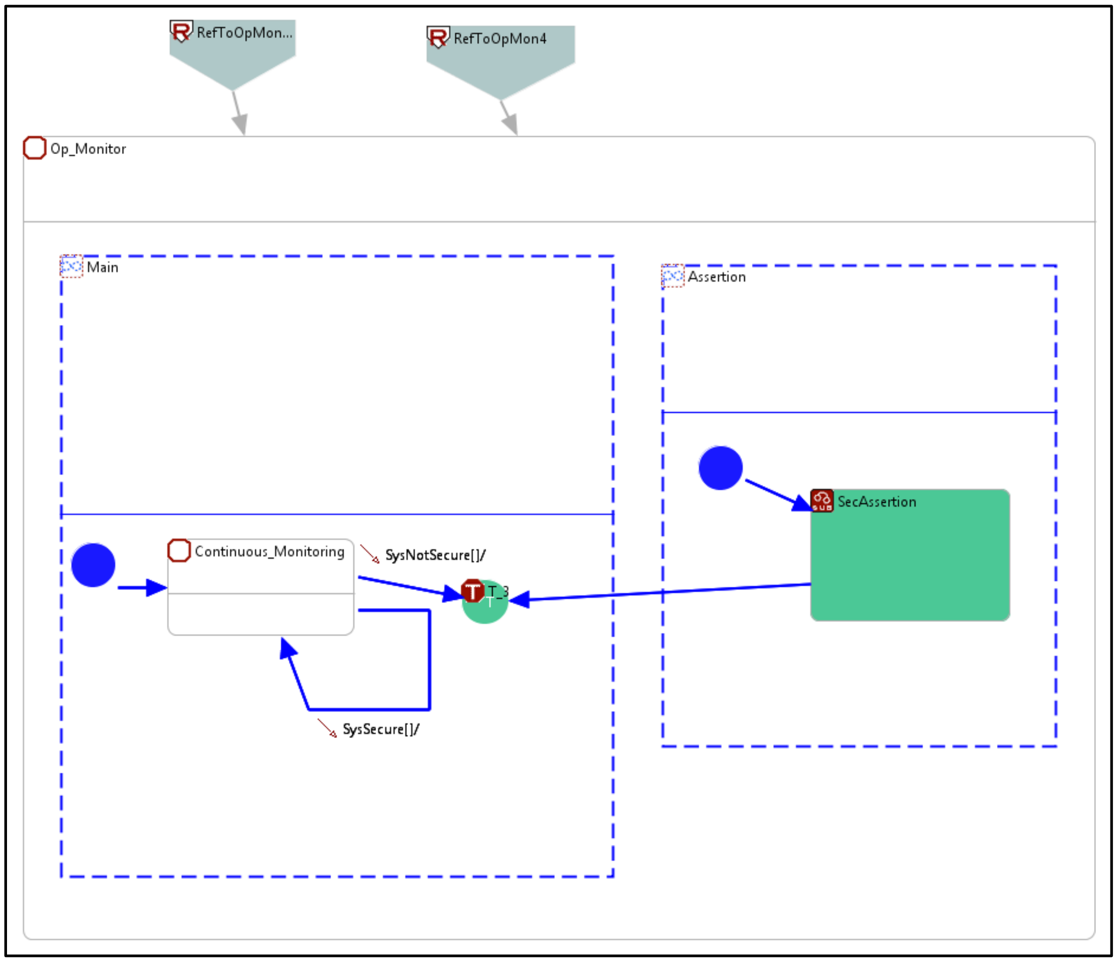


Figure 66. State *Op_Monitor* with Embedded Statechart Assertion

6. V&V Process Model

a. Validation

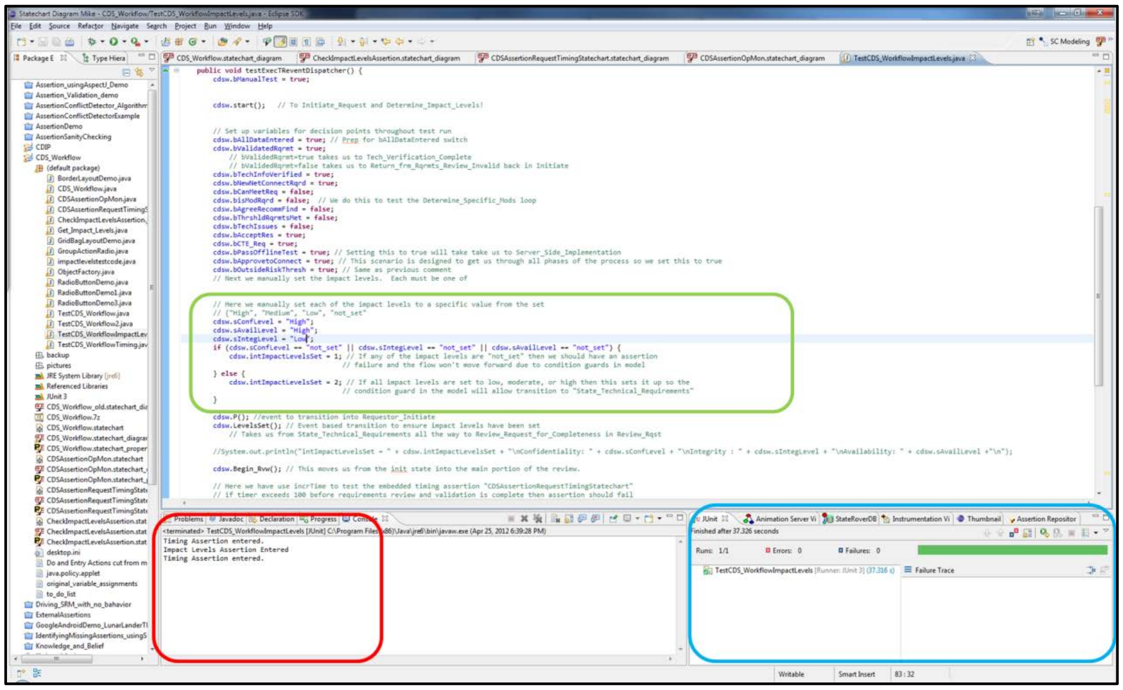
The process engineer presents the finalized model to process stakeholders to ensure the model meets the expectations and requirements of stakeholders. This interaction is depicted in Figure 6 by the dashed line from *V&V Process* to *Process Stakeholder Expectations*. It represents a key portion of the feedback loop with stakeholders and facilitates validation of the process model. For the purposes of demonstrating the feasibility of this component of our modeling approach, we worked directly with UCDMO to ensure the completed process model met their expectations.

We presented our model to the UCDMO in a similar fashion to what we described in Section IV.A.6.a for the CDIP. We informally presented diagrams of the formal model and discussed the translation process from UCDMO supplied process documentation to the process model. The resulting dialogue provided validation that our design met stakeholder intent. We used this feedback loop to ensure that the “Operate and Monitor” component of the model, mentioned in Section IV.A.2 met the UCDMO expectations. Due to the lack of documentation available for analysis of “Operate and Monitor” it was particularly helpful to have direct input from the process stakeholders to improve the process model.

b. Verification – Manual Testing

We use the same process for manual testing as described for the CDIP in Section IV.A.6.b of this document. Manual testing allows the process engineer to focus on and test specific scenarios to examine how a given set of variable values, timings, and events will affect the model during execution. In this section we use manually generated test cases to demonstrate how statechart assertions embedded in the CDS Workflow enforce requirements on the process model.

The test case shown in Figure 67 is designed to test the assertion *CheckImpactLevelsAssertion* to ensure that the assertion does not fail when it should succeed. This assertion, shown in Figure 61 is designed enforce the natural language requirement R1: *Each of the impact levels (Confidentiality, Integrity, Availability) must be set to one of the following: low, moderate, high*. In the green outlined area of Figure 67, we assign values of High, High, and Low to the Confidentiality, Availability, and Integrity impact levels, respectively. In the red outlined area of Figure 67, the message “Impact Levels Assertion Entered” shows that the assertion was entered while the blue outlined area shows that no assertions failed during the test run.



```
public void testExecTReventDispatcher() {
    cdsw.bManualTest = true;


    cdsw.start();   // To Initiate_Request and Determine_Impact_Levels!


    // Set up variables for decision points throughout test run
    cdsw.bAllDataEntered = true; // Prep for bAllDataEntered switch
    cdsw.bValidatedRqrmt = true;
        // bValidedRqrmt=true takes us to Tech_Verification_Complete
        // bValidedRqrmt=false takes us to Return_frm_Rqrmts_Review_Invalid back in Initiate
    cdsw.bTechInfoVerified = true;
    cdsw.bNewNetConnectRqrd = true;
    cdsw.bCanMeetReq = false;
    cdsw.bisModRqrd = false;  // We do this to test the Determine_Specific_Mods loop
    cdsw.bAgreeRecommFind = false;
    cdsw.bThrshldRqrmtsMet = false;
    cdsw.bTechIssues = false;
    cdsw.bAcceptRes = true;
    cdsw.bCTE_Req = true;
    cdsw.bPassOfflineTest = true; // Setting this to true will take take us to Server_Side_Implementation
    cdsw.bApprovetoConnect = true; // This scenario is designed to get us through all phases of the process so we set this to true
    cdsw.bOutsideRiskThresh = true; // Same as previous comment
    // Next we manually set the impact levels.  Each must be one of

    // Here we manually set each of the impact levels to a specific value from the set
    // {"High", "Medium", "Low", "not_set"
    cdsw.sConfLevel = "High";
    cdsw.sAvailLevel = "High";
    cdsw.sIntegLevel = "Low";
    if (cdsw.sConfLevel == "not_set" || cdsw.sIntegLevel == "not_set" || cdsw.sAvailLevel == "not_set") {
        cdsw.intImpactLevelsSet = 1; // If any of the impact levels are "not_set" then we should have an assertion
                                     // failure and the flow won't move forward due to condition guards in model
    } else {
        cdsw.intImpactLevelsSet = 2; // If all impact levels are set to low, moderate, or high then this sets it up so the
                                     // condition guard in the model will allow transition to "State_Technical_Requirements"
    }
    cdsw.P(); //event to transition into Requestor_Initiate
    cdsw.LevelsSet(); // Event based transition to ensure impact levels have been set
        // Takes us from State_Technical_Requirements all the way to Review_Request_for_Completeness in Review_Rqst

    //System.out.println("intImpactLevelsSet = " + cdsw.intImpactLevelsSet + "\nConfidentiality: " + cdsw.sConfLevel + "\nIntegrity : " + cdsw.sIntegLevel + "\nAvailability: " + cdsw.sAvailLevel +"\n");

    cdsw.Begin_Rvw(); // This moves us from the init state into the main portion of the review.

    // Here we have use incrTime to test the embedded timing assertion "CDSAssertionRequestTimingStatechart"
    // if timer exceeds 100 before requirements review and validation is complete then assertion should fail
```

```
<terminated> TestCDS_WorkflowImpactLevels [JUnit] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Apr 25, 2012 6:39:28 PM)
Timing Assertion entered.
Impact Levels Assertion Entered
Timing Assertion entered.
```

```
Finished after 37.326 seconds
Runs: 1/1     Errors: 0     Failures: 0
TestCDS_WorkflowImpactLevels [Runner: JUnit 3] (37.316 s)
```

Figure 67. Successful Manual Test of *CheckImpactLevelsAssertion*

For those test runs with animation activated, our modeling tool provides additional visual feedback as shown in Figure 68. In this case the assertion's failure criteria were not met so the statechart assertion remained in the *Start* state throughout the test run.

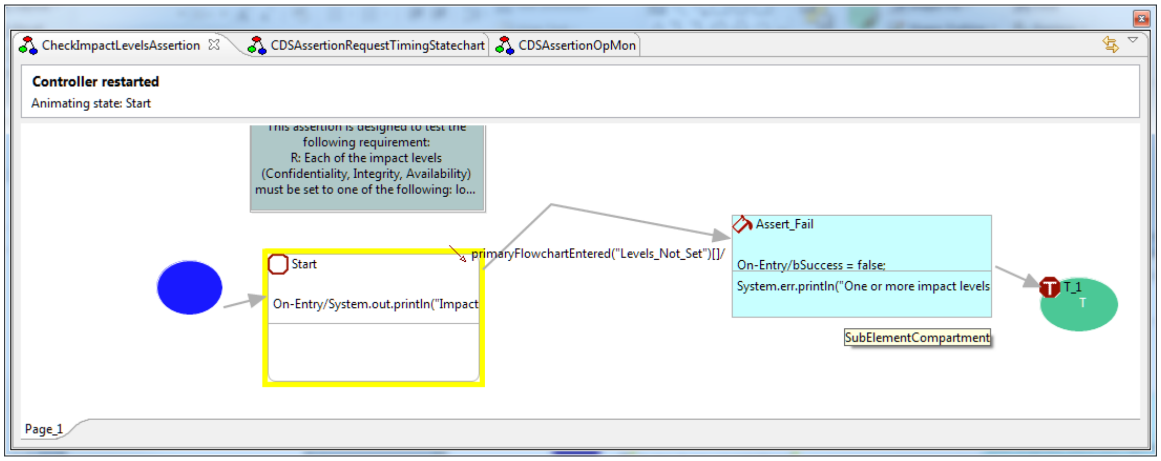
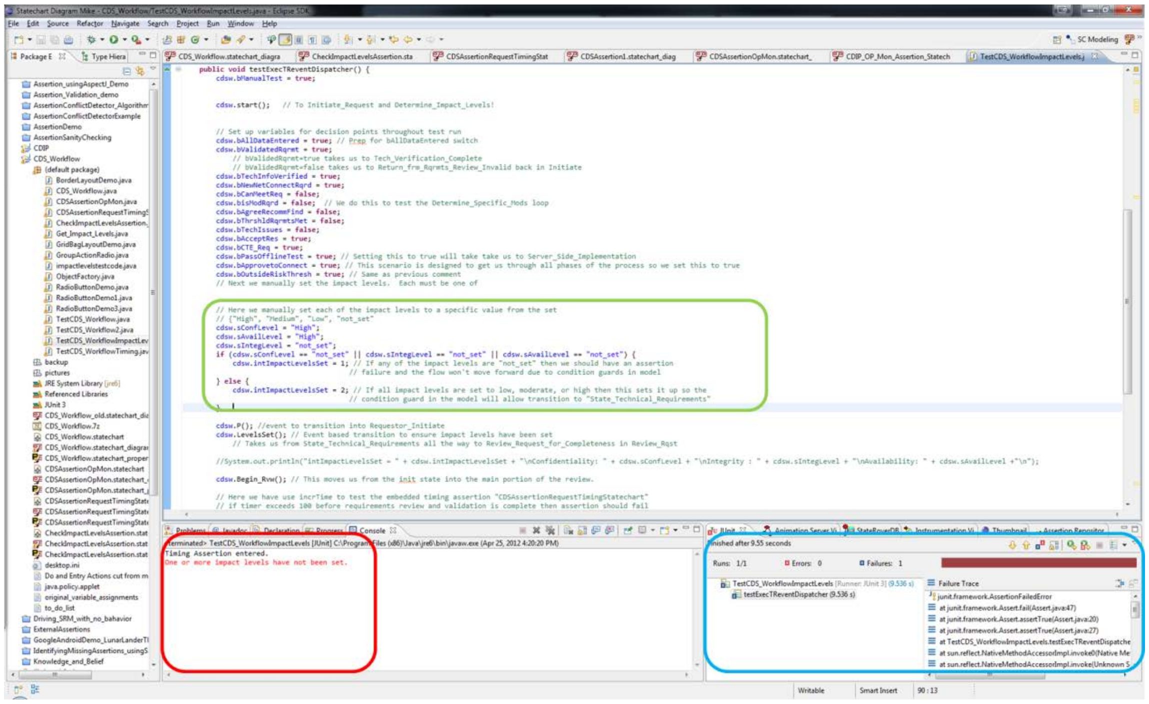

CheckImpactLevelsAssertion
CDSAssertionRequestTimingStatechart
CDSAssertionOpMon
Controller restarted
Animating state: Start
This assertion is designed to test the following requirement:
R: Each of the impact levels (Confidentiality, Integrity, Availability) must be set to one of the following: lo...
Start
On-Entry/System.out.println("Impact
primaryFlowchartEntered("Levels_Not_Set")[]/
Assert_Fail
On-Entry/bSuccess = false;
System.err.println("One or more impact levels
SubElementCompartment
T_1
T
Page_1

Figure 68. Assertion *CheckImpactLevelsAssertion* Successful Test Run

The next test results show the indicators for a failed test run. In Figure 69, the green outlined area shows that we set the Integrity impact level to "not_set." This caused the *CheckImpactLevelsAssertion* embedded assertion to fail and print the status message "One or more impact levels have not been set." in the red outlined area and to throw an *AssertionFailedError* in the blue outlined area of Figure 69.



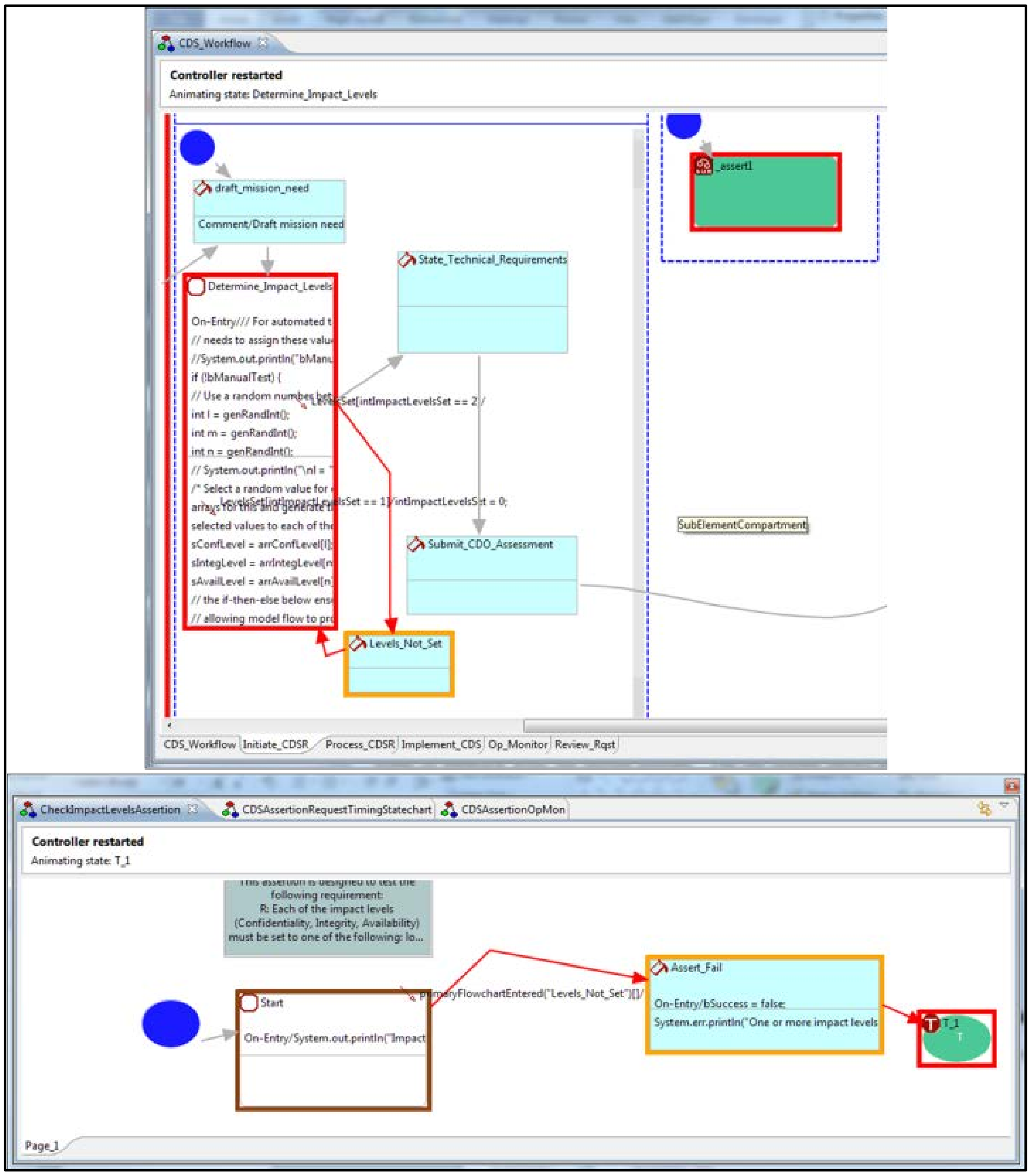
```
    public void testExecTReventDispatcher() {
        cdsw.bManualTest = true;


        cdsw.start();   // To Initiate_Request and Determine_Impact_Levels!


        // Set up variables for decision points throughout test run
        cdsw.bAllDataEntered = true; // Prep for bAllDataEntered switch
        cdsw.bValidatedRqrmt = true;
            // bValidedRqrmt=true takes us to Tech_Verification_Complete
            // bValidedRqrmt=false takes us to Return_frm_Rqrmts_Review_Invalid back in Initiate
        cdsw.bTechInfoVerified = true;
        cdsw.bNewNetConnectRqrd = true;
        cdsw.bCanMeetReq = false;
        cdsw.bisModRqrd = false;  // We do this to test the Determine_Specific_Mods loop
        cdsw.bAgreeRecommFind = false;
        cdsw.bThrshldRqrmtsMet = false;
        cdsw.bTechIssues = false;
        cdsw.bAcceptRes = true;
        cdsw.bCTE_Req = true;
        cdsw.bPassOfflineTest = true; // Setting this to true will take take us to Server_Side_Implementation
        cdsw.bApprovetoConnect = true; // This scenario is designed to get us through all phases of the process so we set this to true
        cdsw.bOutsideRiskThresh = true; // Same as previous comment
        // Next we manually set the impact levels.  Each must be one of

        // Here we manually set each of the impact levels to a specific value from the set
        // {"High", "Medium", "Low", "not_set"
        cdsw.sConfLevel = "High";
        cdsw.sAvailLevel = "High";
        cdsw.sIntegLevel = "not_set";
        if (cdsw.sConfLevel == "not_set" || cdsw.sIntegLevel == "not_set" || cdsw.sAvailLevel == "not_set") {
            cdsw.intImpactLevelsSet = 1; // If any of the impact levels are "not_set" then we should have an assertion
                                         // failure and the flow won't move forward due to condition guards in model
        } else {
            cdsw.intImpactLevelsSet = 2; // If all impact levels are set to low, moderate, or high then this sets it up so the
                                         // condition guard in the model will allow transition to "State_Technical_Requirements"
        }

        cdsw.P(); //event to transition into Requestor_Initiate
        cdsw.LevelsSet(); // Event based transition to ensure impact levels have been set
            // Takes us from State_Technical_Requirements all the way to Review_Request_for_Completeness in Review_Rqst

        //System.out.println("intImpactLevelsSet = " + cdsw.intImpactLevelsSet + "\nConfidentiality: " + cdsw.sConfLevel + "\nIntegrity : " + cdsw.sIntegLevel + "\nAvailability: " + cdsw.sAvailLevel +"\n");

        cdsw.Begin_Rvw(); // This moves us from the init state into the main portion of the review.

        // Here we have use incrTime to test the embedded timing assertion "CDSAssertionRequestTimingStatechart"
        // if timer exceeds 100 before requirements review and validation is complete then assertion should fail
```

```
<terminated> TestCDS_WorkflowImpactLevels [JUnit] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Apr 25, 2012 4:20:20 PM)
Timing Assertion entered.
One or more impact levels have not been set.
```

```
Finished after 9.55 seconds
Runs: 1/1    Errors: 0    Failures: 1

TestCDS_WorkflowImpactLevels [Runner: JUnit 3] (9.536 s)
    testExecTReventDispatcher (9.536 s)

Failure Trace
junit.framework.AssertionFailedError
at junit.framework.Assert.fail(Assert.java:47)
at junit.framework.Assert.assertTrue(Assert.java:20)
at junit.framework.Assert.assertTrue(Assert.java:27)
at TestCDS_WorkflowImpactLevels.testExecTReventDispatche
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Me
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown S
```

Figure 69. Failed Manual Test of CheckImpactLevelsAssertion

In the lower portion of Figure 70, we see a transition from the *Start* state to the *Assert_Fail* flowchart box and on to a terminal state. This occurred because one of the impact levels was not set in the CDS Workflow model, causing a transition to the flowchart box *Levels_Not_Set* which is shown by the orange outlined box in the upper portion of Figure 70.



Controller restarted
Animating state: Determine_Impact_Levels

```
Comment/Draft mission need
```

```
On-Entry/// For automated t
// needs to assign these valu
//System.out.println("bManu
if (!bManualTest) {
// Use a random numbers bet
int l = genRandInt();
int m = genRandInt();
int n = genRandInt();
// System.out.println("\nl = "
/* Select a random value for
arrays for this and generate t
selected values to each of the
sConfLevel = arrConfLevel[l];
sIntegLevel = arrIntegLevel[m
sAvailLevel = arrAvailLevel[n]
// the if-then-else below ens
// allowing model flow to pr
```

```
LevelsSet[intImpactLevelsSet == 2]/
LevelsSet[intImpactLevelsSet == 1]/intImpactLevelsSet = 0;
```

Controller restarted
Animating state: T_1

```
This assertion is designed to test the
following requirement:
R: Each of the impact levels
(Confidentiality, Integrity, Availability)
must be set to one of the following: lo...
```

```
On-Entry/System.out.println("Impact
```

```
primaryFlowchartEntered("Levels_Not_Set")[]/
```

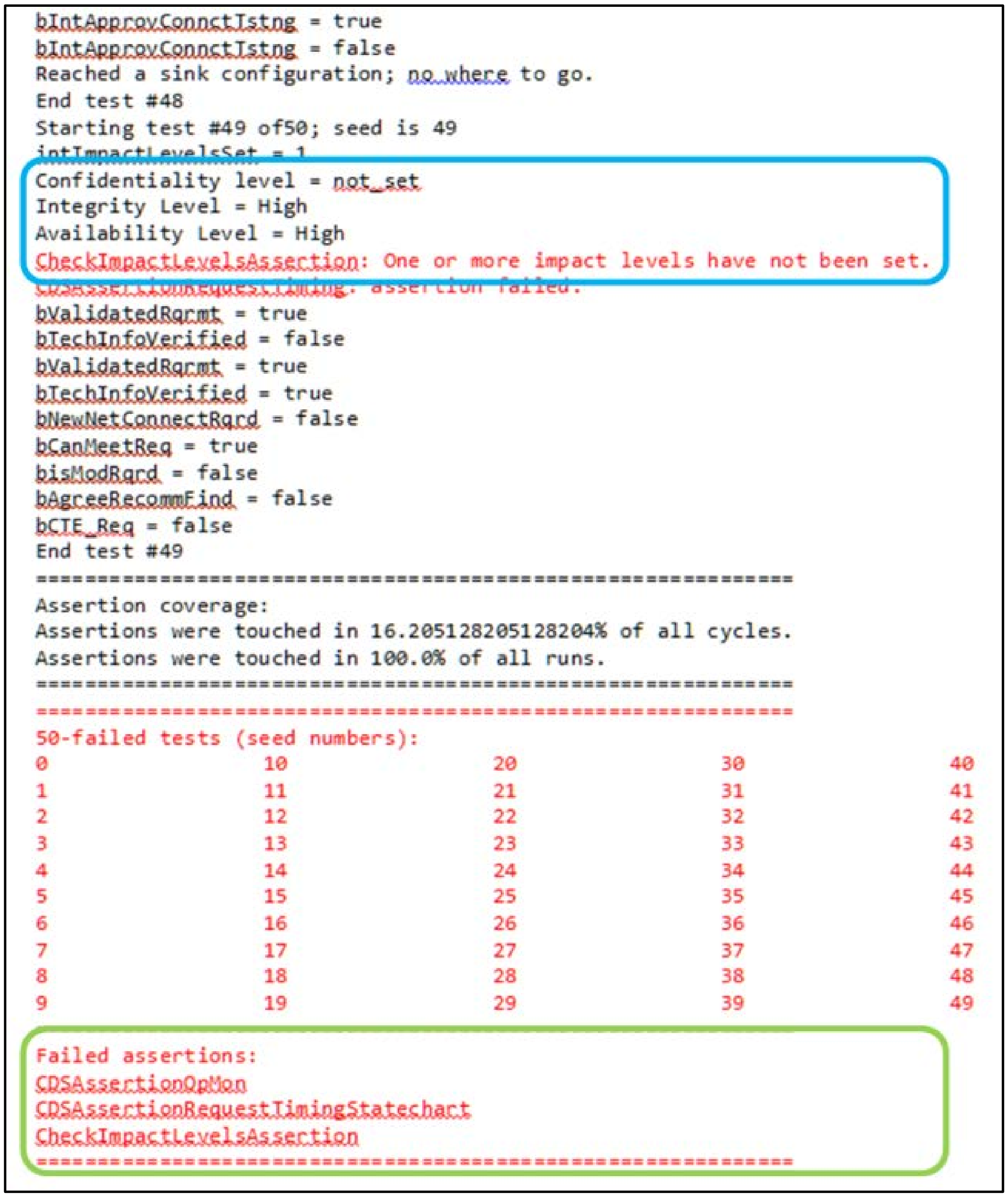
```
On-Entry/bSuccess = false;
System.err.println("One or more impact levels
```

Figure 70. Assertion CheckImpactLevelsAssertion Failed Test Run

When we write manual test cases, we are able to test process flow and adherence to process requirement written as embedded assertions by specifying events and setting variables to move through the executing process in a specific way. This ensures that the model behaves as expected for each test scenario.

c. Verification – Runtime Execution Monitoring

In this section we show the results of automated testing of the CDS Workflow process model. Runtime execution monitoring provides a means of exploring the effect of numerous input sequences on the process model during automated testing and verifying that the model behaves as expected across the range of inputs. During model construction we insert code to print variable values and status messages at runtime. These messages are delivered throughout execution of each test. This helps us ensure that the model behaves precisely as specified since we are able to compare variable values for a test run to the testing results based on those values. An example of this can be seen in the blue outline of Figure 71. The Confidentiality impact level is “not_set” which should cause the assertion *CheckImpactLevelsAssertion* to fail and it does, as we see from the status message in red “CheckImpactLevelAssertion: One or more impact levels have not been set.”



```
bIntApprovConnctIstng = true
bIntApprovConnctIstng = false
Reached a sink configuration; no where to go.
End test #48
Starting test #49 of50; seed is 49
intImpactLevelsSet = 1
Confidentiality level = not_set
Integrity Level = High
Availability Level = High
CheckImpactLevelsAssertion: One or more impact levels have not been set.
CDSAssertionRequestTiming: assertion failed.
bValidatedRqrmt = true
bTechInfoVerified = false
bValidatedRqrmt = true
bTechInfoVerified = true
bNewNetConnectRqrd = false
bCanMeetReq = true
bisModRqrd = false
bAgreeRecommFind = false
bCTE_Req = false
End test #49
============================================================
Assertion coverage:
Assertions were touched in 16.205128205128204% of all cycles.
Assertions were touched in 100.0% of all runs.
============================================================
============================================================
50-failed tests (seed numbers):
0                  10                 20                 30                 40
1                  11                 21                 31                 41
2                  12                 22                 32                 42
3                  13                 23                 33                 43
4                  14                 24                 34                 44
5                  15                 25                 35                 45
6                  16                 26                 36                 46
7                  17                 27                 37                 47
8                  18                 28                 38                 48
9                  19                 29                 39                 49

Failed assertions:
CDSAssertionOpMon
CDSAssertionRequestTimingStatechart
CheckImpactLevelsAssertion
============================================================
```

Figure 71. WTG Output for 50-Test Run

Depending on our testing goals and the outcome of test runs, we adjust test parameters such as number of tests, test length, number of permissible loops in order to ensure all elements that we wish to examine have been tested. The green outline of Figure 71 shows us that all three of the assertions in our model failed during this 50-test run. We expect this to occur as the WTG explores the execution paths for a large number of input sequences over the course of fifty tests.

C. KEY LESSONS LEARNED

As we developed the CDIP and CDS Workflow models there were some key lessons learned that will facilitate the improvement of further research into the modeling of partially automated human-based security-related processes.

1. Code Generation

The code generation facility turned out to be a key means of ensuring end-to-end syntactic and semantic consistency of our model. Note that during model development we do not focus on code generation from the standpoint of creating usable code as would be the case when developing software-based systems. Instead, we are interested in the rigorous syntax and semantics checking that is an inherent part of StateRover’s code generation module. If code generation is unable to complete successfully this equates to a syntactic or semantic inconsistency with the model and drives the process engineer to investigate and correct the source of the error before continuing with model development.

2. Source Material

During the “Process Analysis” phase of our modeling approach, the process engineer develops an understanding of the process under examination based on the materials provided by the stakeholders of the process. This material may be abundant or scarce and could include things such as interviews with users or stakeholders, focus groups, informal diagrams, or semi-formal diagrams. We modeled two processes during the course of this research. The source material differed significantly between the two processes and we found in the case of the CDS Workflow that having the UCDMO provided activity diagrams and use cases facilitated our understanding of model.

If provided a similar set of activity diagrams for any process, it would facilitate an expeditious analysis of the process. However, the notion of “garbage in, garbage out” applies here. In other words, if it is the case that either the process diagrams or documentation is inaccurate or incomplete, we believe that analysis could end up taking longer or result in unfounded conclusions due to the original diagrams leading the analysis down one or more false paths. Process diagrams are not a requirement for analysis but instead a facilitator. In the absence of diagrams or other forms of documentation, the process engineer engaged in formalizing a process is likely to employ a variety of methods to develop a full understanding of the process such as observation of the process in action, and interviews with stakeholders. Based on our experience with the two processes modeled in this work, we suggest that inaccurate, inconsistent, or non-existent process documentation would significantly increase the process-analysis timeline due to the need for end-to end-process analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

A. CONCLUSIONS

In this research, we demonstrated the development of a systematic approach to formally modeling human-in-the-loop security analysis and decision-making processes as well as the use of UML statecharts and statechart assertions for engineering, modeling, and V&V of these processes. The contributions described below support this research.

1. Software Engineering

We contributed to software engineering by introducing a novel way for software to automate a new domain, that being the process-modeling engineering of high-level, human-based processes. We do this through the generation of an executable process model and executable assertions. We then use software to enact these executable representations as a means of process automation.

2. Process-modeling Engineering

We developed a systematic approach to formally modeling, validating and verifying high-level human-based processes (shown in Figure 72). These processes can be challenging to model because of hard-to-capture elements such as human decision-making, sequencing, and concurrent activities. We applied some of the tools and techniques from software engineering to provide an end-to-end means of modeling and V&V of these processes using the same formalism. This provides a framework for the specification of security processes and computer-assisted V&V of the specifications.

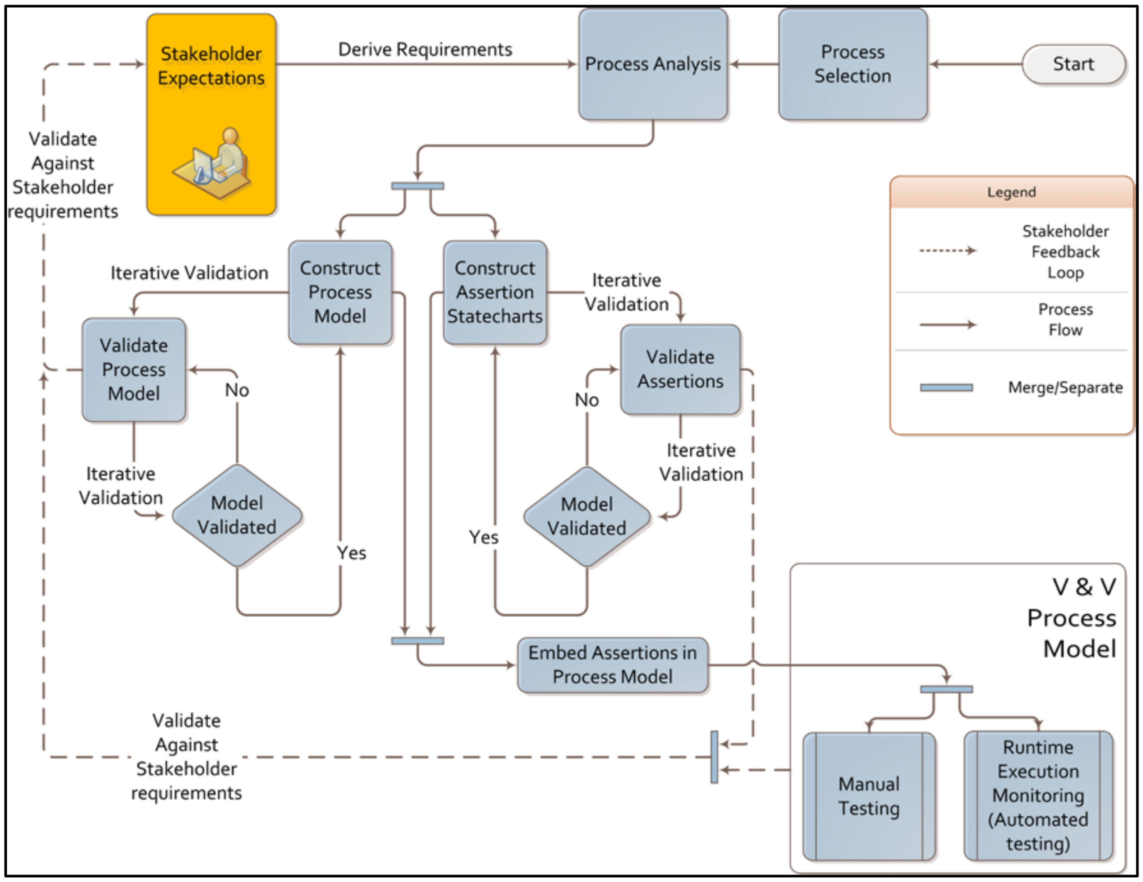


Figure 72. Statechart-Based Formal Modeling Approach

We developed a set of desirable attributes to guide the choice of a formal language and associated computer-based tools that would support our modeling approach. We examined a number of formal languages and tools in the context of these attributes and showed that UML statecharts and statechart assertions in conjunction with the tools available for this formal language provide us with the necessary vehicle for building a formal process model as well as specifying and enforcing requirements on the model.

Our approach includes development of an executable version of the modeled process and the process requirements models as statechart assertions. Once developed, this executable model provides us with a runtime view of the process. We use formal methods tools and techniques originally designed for the engineering of reactive hardware and software systems as a means to monitor the process in execution for requirements satisfaction. The addition of runtime monitoring, in conjunction with embedded statechart-based assertions, offer the process engineer an unprecedented ability to levy requirements on a human-based process and enforce those requirements while the process is in execution.

3. Case Studies

We demonstrate the application of our systematic modeling approach through two case studies. These two cases represent hard process modeling problems and encompass a large portion of the hard-to-capture elements mentioned above. By automating the process engineering we can capture and V&V the processes. The two case studies are based on real-world processes used by the UCDMO for the development, implementation, and C&A of CDS.

4. Real-world Impact

We provided feedback to the UCDMO on process errors discovered through the case studies, resulting in corresponding changes to the real-world process for requesting, developing, implementing, certifying and accrediting cross domain solutions. This demonstrates how the embedded feedback loop in the process modeling approach can directly contribute to process engineering and improvement of real-world processes.

B. RECOMMENDATIONS FOR FUTURE WORK

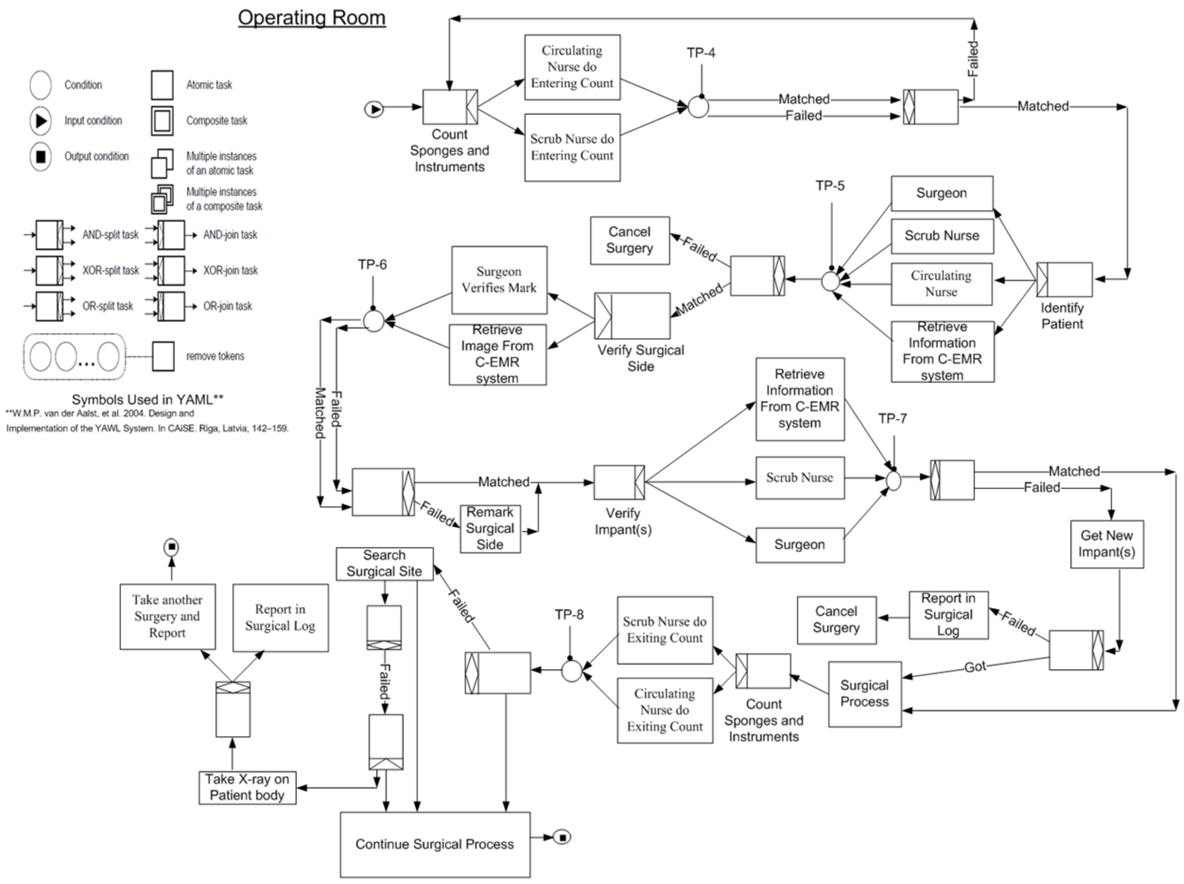
More work is needed to further validate our modeling approach. We have applied our approach to two processes in the security analysis and decision-making domain and submit that it would improve process-modeling efforts in other domains. In addition, there needs to be additional research to enable the long-term runtime monitoring of an executable process model in direct support of the real-world application of the process. In other words, it would be desirable to provide the users of the approach with the capability to execute process models for sufficiently long periods of time such that the modeled processes terminate naturally (i.e., processes that have a definitive end-point or product which causes the process to end) or run indefinitely (e.g., safety processes that involve continuous checking of health and status of the manual and automated functions of a process-control application). Future research is needed on the optimal placement and use of embedded assertions within a statechart-based formal process model. Follow-on research also needs to examine additional modeling tool capabilities to facilitate the modeling of human-in-the-loop processes.

1. Improving Workflows for Surgical Procedures

In this research, we apply the modeling approach to partially automated, human-based, C&A processes. There are also opportunities in the field of medical workflow specifications for further validation of our modeling approach.

Since 1993, the DoD has transformed health care delivery in its use of information technology to automate patient data documentation. The Department uses an enterprise-wide medical and dental clinical information system that generates, maintains, and provides 24-hour secure online access to electronic medical records (EMR). This system of EMRs enhances patient safety for more than nine million beneficiaries, with "one patient, one record." It provides a legible and longitudinal clinical record that includes drug interaction alerts, patient allergy notifications, and wellness reminders to enhance health care delivery. (Charles, Harmon, and Jordan 2005)

Under the rubric of the Military Health System (MHS), DoD operates state-of-the-art hospitals and clinics, battlefield, and forward-deployed temporary medical facilities worldwide. MHS provides care to over 19,000 inpatients and 1.7 million outpatients each week (Charles, Harmon, and Jordan 2005). The EMR is a primary enabler for the improvement in safety, effectiveness, and efficiency of healthcare and as a fully integrated component of the military healthcare paradigm. Through the integration of Health Information Technology (HIT) such as the EMR, the DoD is searching for ways to improve the quality and efficiency of care it provides to members of the military. In addition to improving EMRs, the DoD has specified, using natural language and simple flowcharts, the process workflows for performing medical procedures. These procedures, such as surgeries, can be performed manually or semi-automatically and may involve both human decision-making and robotically controlled elements.



Operating Room
Condition
Input condition
Output condition
Atomic task
Composite task
Multiple instances of an atomic task
Multiple instances of a composite task
AND-split task
AND-join task
XOR-split task
XOR-join task
OR-split task
OR-join task
remove tokens
Symbols Used in YAML**
**W.M.P. van der Aalst, et al. 2004. Design and Implementation of the YAWL System. In CAiSE. Riga, Latvia, 142–159.
Count Sponges and Instruments
Circulating Nurse do Entering Count
Scrub Nurse do Entering Count
TP-4
Matched
Failed
TP-5
Surgeon
Scrub Nurse
Circulating Nurse
Retrieve Information From C-EMR system
Identify Patient
Cancel Surgery
TP-6
Surgeon Verifies Mark
Retrieve Image From C-EMR system
Verify Surgical Side
Remark Surgical Side
Verify Impant(s)
Scrub Nurse
Surgeon
TP-7
Get New Impant(s)
Report in Surgical Log
Surgical Process
Got
TP-8
Scrub Nurse do Exiting Count
Circulating Nurse do Exiting Count
Search Surgical Site
Take another Surgery and Report
Report in Surgical Log
Take X-ray on Patient body
Continue Surgical Process

Figure 73. Example Surgical Procedure Workflow (From Yu et al. 2011)

Yu, Varga, Wijesekera, Stavrou, and Singhal are investigating the improvement of surgical procedure workflows (example shown in Figure 73) and surgical electronic medical records (S-EMR) through the application of use/misuse cases and time-out points. They describe workflows for surgical procedures that incorporate both EMRs and more pedestrian means such as paper-based checklists. They discuss the inclusion of time-out points as a means of reducing injury and casualty rates during surgical procedures. Time-out points provide a controlled pause in the medical procedure, affording the surgical team time to check a pre-defined condition before proceeding to the next step. They propose the architecture shown in Figure 74 for enforcing time-out points in workflows as a means of ensuring the rigorous application of time-out points throughout the procedure. (Yu et al. 2011).

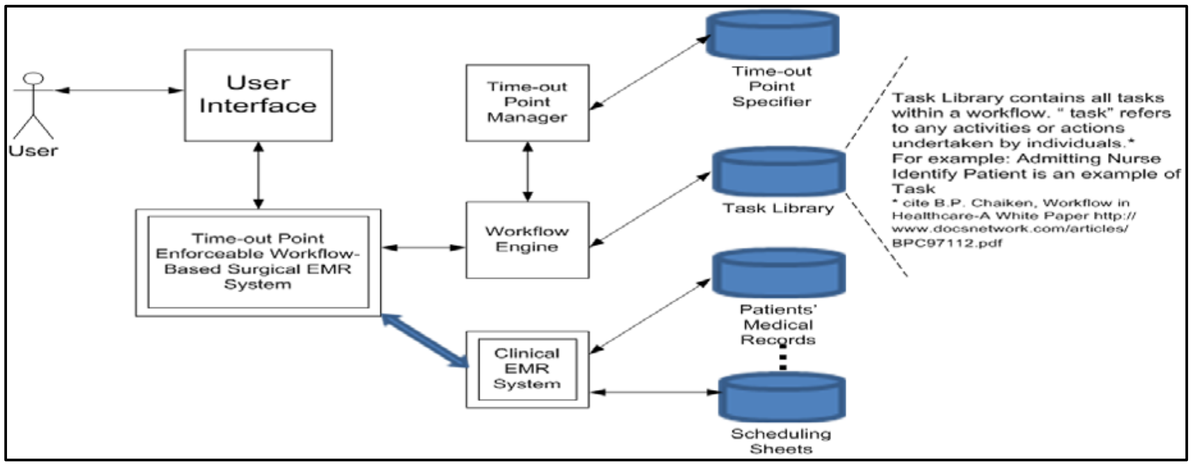

User
User Interface
Time-out Point Manager
Time-out Point Specifier
Task Library contains all tasks within a workflow. " task" refers to any activities or actions undertaken by individuals.* For example: Admitting Nurse Identify Patient is an example of Task
* cite B.P. Chaiken, Workflow in Healthcare-A White Paper http:// www.docsnetwork.com/articles/ BPC97112.pdf
Task Library
Time-out Point Enforceable Workflow-Based Surgical EMR System
Workflow Engine
Patients' Medical Records
Clinical EMR System
Scheduling Sheets

Figure 74. Time-out Point Enforceable Architecture (From Yu et al. 2011)

Improving safety and reducing error rates in medical procedures is an important area of research to DoD and to the medical community at large. We propose that future researchers use DoD medical workflow specifications to further validate our approach to applying computer-aided formal V&V of process workflows. We also propose that the runtime execution monitoring of embedded assertions, used as a requirements-enforcement mechanism in our modeling approach, could be used as the basis for a Time-out Point Manager (TPM).

2. Runtime Execution Monitoring – Long-term Approach

We demonstrated a process modeling approach that supports V&V of the modeled process through runtime execution monitoring and the enforcement of embedded assertions. Further work is needed to support the monitoring and enforcement of process requirements throughout a process lifecycle. We believe the approach described in this research could form the basis for an enforcement engine that ensures a real-world manifestation of the modeled process adheres to process requirements. The TPM described in the previous section is an example of a requirements enforcement mechanism that would operate in conjunction with the real-world execution of a process.

3. Full-scale Employment of Embedded Assertions

We demonstrated the use of several embedded statechart assertions to model and enforce process requirements. We believe that many modeled processes would have a large number of stakeholder requirements, translating to a large number of embedded assertions in the process model. Future research needs to further examine full-scale employment of embedded assertions to model and enforce all of these process requirements. In a full-scale deployment, it would be important to determine *a priori* whether the enforcement of embedded assertions introduces a performance penalty and if so, how much of one. We did not see any appreciable runtime performance penalties when adding statechart assertions to our models; however, it may be the case that working with a large number of assertions would deteriorate runtime execution monitoring performance.

4. Validation Using External Assertions

We explored the use of embedded assertions statecharts as a requirements enforcement mechanism on models of partially automated human-based security-related processes. Embedded statecharts assertions, as the name implies, monitor and enforce "from the inside" of the modeled process. This approach was appropriate to our research since we modeled both the assertions and process models in the same language, using the same toolset.

Future research should investigate using external assertions and assertion repositories as a means of monitoring "from the outside" of the modeled process. Drusinky introduced the concepts and provided usage examples of external assertion repositories (Drusinsky et al. 2008; Drusinsky 2011, 58–79). This approach would likely enable the use of differing tools, techniques, and languages for the process model and the assertions used to enforce requirements on the model.

5. Additional Modeling Tool Capabilities

The StateRover modeling tool used in this research satisfied the list of minimum desirable attributes detailed in Section III.A.1. Future research needs to examine continued development of tools to support the modeling of decision-making processes. In Section III.C.6.c we discussed data collection and analysis related to runtime execution modeling. It would have been helpful to have a robust, automated data collection and reporting mechanism built into the modeling tool to collect data such as the length of time for each test run, number of assertions that passed and the number that failed in each test run, and the number of model states not visited. We manually collected this data and used it to help with the development and debugging of process models.

LIST OF REFERENCES

Ambler, Scott W. 2005. *The Elements of UML(TM) 2.0 Style*. Cambridge University Press.

Anon. *Jobs At Netflix, Inc.* http://www.netflix.com/Jobs.

Berry, D M, & J M Wing. 1985. “Specifying and Prototyping: Some Thoughts on Why They Are Successful.” In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software, Vol.2: Colloquium on Software Engineering (CSE)*, 117–128. New York, NY, USA: Springer-Verlag New York, Inc. http://dl.acm.org/citation.cfm?id=22263.22271.

Bishop, Matt. 2002. *Computer Security: Art and Science*. 1st ed. Addison-Wesley Professional.

Bruegge, Bernd, & Allen H. Dutoit. 2004. *Object-oriented software engineering: using UML, patterns and Java*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Cardoso, Jorge. 2007. “Complexity Analysis of BPEL Web Processes.” *Software Process: Improvement and Practice* 12 (1): 35–49. doi:10.1002/spip.302.

Cardoso, J. Mendling, G. Neumann, & H A Reijers. 2006. “A Discourse on Complexity of Process Models (Survey Paper).” *BPM 2006 Workshops LNCS 4103* 4103: 115–126.

Charles, Marie-Jocelyne, Bart J Harmon, & Pamela S Jordan. 2005. *Improving Patient Safety With the Military Electronic Health Record*. Rockville, MD: Agency for Healthcare Research and Quality. http://www.dtic.mil/docs/citations/ADA434219.

Clempner, Julio. 2010. “A Hierarchical Decomposition of Decision Process Petri Nets for Modeling Complex Systems.” *International Journal of Applied Mathematics and Computer Science* 20 (2) (June 1): 349–366. doi:10.2478/v10006-010-0026-2.

Committee on National Security Systems. 2006. *National Information Assurance Glossary*. http://www.cnss.gov/full-index.html.

Crane, Michelle, and Juergen Dingel. 2007. “UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal.” *Software and Systems Modeling* 6 (4): 415–435.

Denning, Peter J. 1971. “Third Generation Computer Systems.” *ACM Computing Surveys* 3 (4): 175–216.

Van Der Aalst, W. M., & Kees Max Van Hee. 2004. *Workflow Management: Models, Methods, and Systems*. MIT Press.

Director of National Intelligence. 2008. *Intelligence Community Directive Number 503: Intelligence Community Information Technology Systems Security Risk Management Certification and Accreditation*. https://www.intelink.gov/sites/ucdmo.

Dong, Yang, & Zhang Shensheng. 2003. *Modeling Workflow Process Models with Statechart*.

Van Dongen, B. F., W. M. P. Van Der Aalst, & H. M. W. Verbeek. 2005. “Verification of EPCs: Using Reduction Rules and Petri Nets.” In *17th International Conference on Advanced Information Systems Engineering*, 3520:372–386. Porto, Portugal.

Van Dongen, B. F., & M. H. Jansen-Vullers. 2005. “Verification of SAP Reference Models.” In *3rd Internaional Conference on Business Process Management*, 3649:464–469. Nancy, France.

Drusinsky, D, M Shing, & K. A. Demir. 2007. “Creating and Validating Embedded Assertion Statecharts.” *Distributed Systems Online, IEEE* 8 (5): 3–3.

Drusinsky, D. 2006. *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*. Newnes. http://isbndb.com/d/book/modeling_and_verification_using_uml_statecharts.

———. 2008. “From UML Activity Diagrams to Specification Requirements.” In *Proceedings of the 2008 IEEE International Conference on System of Systems Engineering*. Monterey, CA.

———. 2011. *Practical UML Based Specification, Validation, and Verification of Mission Critical Software*. Dog Ear Publishing, LLC.

Drusinsky, D., J. B. Michael, T. W. Otani, & Man-Tak Shing. 2008. “Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance.” In *Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement, 2008*, 47–51.

Drusinsky, D., James B. Michael, & Mantak Shing. 2007. *The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors*. Naval Postgraduate School.

Drusinsky, D., & Man-Tak Shing. 2009. “Using UML Statecharts with Knowledge Logic Guards.” In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, 586–590. Berlin, Heidelberg: Springer-Verlag. http://dx.doi.org/10.1007/978-3-642-04425-0_45.

Drusinsky, D., Man-Tak Shing, & K. A Demir. 2006. “Creation and Validation of Embedded Assertion Statecharts.” In *Seventeenth IEEE International Workshop on Rapid System Prototyping, 2006*, 17–23. IEEE. doi:10.1109/RSP.2006.12.

Emmerich, W., S. Bandinelli, L. Lavazza, & J. Arlow. 1996. “Fine grained process modelling: an experiment at British Airways.” In *Proceedings of the Fourth International Conference on the Software Process*, 2–12. IEEE. doi:10.1109/ICSP.1996.565016.

Emmerich, W., & V. Gruhn. 1991. “FUNSOFT nets: a Petri-net based software process modeling language.” In *Proceedings of the Sixth International Workshop on Software Specification and Design, 1991*, 175–184. Como, Italy: IEEE. doi:10.1109/IWSSD.1991.213063.

Gabbar, Hossam A. 2006. *Modern Formal Methods and Applications*. Dordrecht: Springer.

Grady, Jeffrey O. 2009. “Universal Architecture Description Framework.” *Systems Engineering* 12 (2): 91–116.

Graham, G. S., & P. J. Denning. 1972. “Protection-principles and Practice.” *AFIPS Conference Proceedings Vol.40, the 1972 Spring Joint Computer Conference*: 417; 417–429; 429.

Gruhn, V. 1992. “Software processes are social processes.” In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering, 1992.*, 196–201. IEEE. doi:10.1109/CASE.1992.200150.

Gruhn, V., & Ralf Laue. 2007. “What Business Process Modelers Can Learn from Programmers.” *Science of Computer Programming* 65 (1) (March): 4–13. doi:10.1016/j.scico.2006.08.003.

Gruhn, Volker, & Ralf Laue. 2006. “Complexity Metrics for Business Process Models.” In *9th International Conference on Business Information Systems (BIS 2006), Volume 85 of Lecture Notes in Informatics*, 1–12.

Hall, Anthony. 2005. “Realising the Benefits of Formal Methods.” In *Formal Methods and Software Engineering*, ed. Kung-Kiu Lau & Richard Banach, 3785:1–4. Berlin, Heidelberg: Springer Berlin Heidelberg. file:///Users/bigschu2/Downloads/citation-1.bib.

Harel, David. 1987. “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming* 8 (3) (June): 231–274. doi:10.1016/0167-6423(87)90035-9.

Hibdon, V.S., & T.C. Hartrum. 1996. “An Air Force Organization Process Model Using Formal Software Engineering Techniques.” In *Proceedings of the National Aerospace and Electronics Conference, 1996.*, 2:482–489. Dayton, OH: IEEE. doi:10.1109/NAECON.1996.517693.

Hoare, C. A. R. 1985. *Communicating sequential processes*. Englewood Cliffs, N.J.: Prentice/Hall International.

Hopcroft, John Edward, Rajeev Motwani, & Jeffrey David Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation*. Boston, MA: Pearson Addison-Wesley.

Hurtado Alegría, Julio A., María Cecilia Bastarrica, & Alexandre Bergel. 2010. *AVISPA: Localizing Improvement Opportunities in Software Process Models*. TR/DCC-2010-6. University of Chile.

———. 2011. “Analyzing Software Process Models with AVISPA.” In *Proceedings of the 2011 International Conference on Software and Systems Process*, 23–32. New York, NY, USA: ACM. doi:10.1145/1987875.1987882.

IAR Systems. 2012. “IAR VisualSTATE.” *IAR Systems: IAR VisualSTATE Component of Embedded Workbench*. http://www.iar.com/en/Products/IAR-visualSTATE/.

Karniel, Arie, & Yoram Reich. “Formalizing a Workflow-Net Implementation of Design-Structure-Matrix-Based Process Planning for New Product Development.” *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 41 (3): 476–491. doi:10.1109/TSMCA.2010.2091954.

Kelley, Dean. 1995. *Automata and Formal Languages□: an Introduction*. Englewood Cliffs, N.J.: Prentice Hall.

Koehler, J., G. Tirenni, & S. Kumaran. 2002. “From Business Process Model to Consistent Implementation: a Case for Formal Verification Methods.” In *Proceedings of the Sixth International Enterprise Distributed Object Computing Conference*, 96–106. Lausanne, Switzerland: IEEE. doi:10.1109/EDOC.2002.1137700.

Lampson, Butler W. 1974. “Protection.” *SIGOPS Operating Systems Review* 8 (1): 18–24.

Michael, J. B., D. Drusinsky, T. W. Otani, & Man-Tak Shing. 2011. “Verification and Validation for Trustworthy Software Systems.” *IEEE Software* 28 (6): 86–92. doi:10.1109/MS.2011.151.

Monin, Jean François, & Michael G Hinchey. 2003. *Understanding formal methods*. Springer.

Muelder, Andreas. 2011. “Yakindu.” *Yakindu Statechart Modeling Tools*. http://www.yakindu.org/yakindu/.

National Institute of Standards and Technology. 2004. *Fips Pub 199: Standards for Security Categorization of Federal Information and Information Systems*. Gaithersburg, MD: National Institute of Standards and Technology.

Niles, K. 2002. *Tribal Knowledge*. Vol. 2008. 6/11/2008.

Oxford English Dictionary. 2012a. “‘Coherence, N.’.” Online Dictionary. *Oxford English Dictionary*. http://www.oed.com/view/Entry/35933?redirectedFrom=coherence.

———. 2012b. “‘Oracle, N.’.” Online Dictionary. *Oxford English Dictionary*. http://www.oed.com/view/Entry/35933?redirectedFrom=coherence.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, & William Lorensen. 1991. *Object-oriented Modeling and Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Ryan, P. Y. A., & S. A. Schneider. 2000. *Modelling and Analysis of Security Protocols*. Addison-Wesley Professional.

Schumann, M. 2009. “A Statechart Model of the Cross Domain Implementation Process.” *Information Assurance Technology Analysis Center Newsletter* 12 (1) (February): 26–30.

Schumann, M., & J.B. Michael. 2009. “Statechart Based Formal Modeling of Workflow Processes.” In *Proceedings of the IEEE International Conference on System of Systems Engineering, 2009*, n.p. Albuquerque, NM.

Sindre, G., & A. L. Opdahl. 2000. “Eliciting Security Requirements by Misuse Cases.” In *37th International Conference on Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings.*, 120.

Sipser, Michael. 1997. *Introduction to the Theory of Computation*. Boston: PWS Pub. Co.

Stuit, Marco. 2011. *Modelling and analysis of human collaboration processes in organization*. Groningen, Netherlands: University of Groningen□; University Library Groningen] [Host].

Unified Cross Domain Management Office. 2008. *Cross Domain Implementation Process*. Unified Cross-Domain Management Office.

———. 2012. “Unified Cross Domain Management Office Portal”. U.S. Government. https://www.intelink.gov/sites/UCDMO/Pages/Default.aspx.

Vlissides, John, Ralph Johnson, & Nick Edgar. 2011. “JUnit: A Cook’s Tour.” *JUnit A Cook’s Tour*. http://junit.sourceforge.net/doc/cookstour/cookstour.htm.

Webster, Noah, & Jean Lyttleton Mckechnie. 1983. *Webster’s New Universal Unabridged Dictionary*. 2nd ed. [S.l.]: Dorset & Baber.

Wong, Peter Y., & Jeremy Gibbons. 2008. “A Process Semantics for BPMN.” In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, 355–374. Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-540-88194-0_22.

Ye, JianHong, ShiXin Sun, Wen Song, and LiJie Wen. 2008. “Formal Semantics of BPMN Process Models Using YAWL.” In *Second International Symposium on Intelligent Information Technology Application*, 2:70–74. Shanghai, China: IEEE. doi:10.1109/IITA.2008.68.

Yu, Bo, J. Varga, Duminda Wijesekera, Angelos Stavrou, & Anoop Singhal. 2011. “Specifying Time-Out Points in Surgical EMR Systems.” In *Proceedings of the International Workshop on Health and Social Care Information Systems*, 221:165–174. Vilamoura, Algarve, Portugal: Springer Berlin Heidelberg.

Zongyan, Qiu, Peng Liyang, & Yang Hongli. 2010. “A Framework for Integrating Human Processes with Business Artifacts.” In *Fifth International Symposium on Service Oriented System Engineering (SOSE)*, 252–259. Nanjing, China: IEEE. doi:10.1109/SOSE.2010.39.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. VADM David Buss
   United States Fleet Forces Command
   Norfolk, VA

4. RADM (S) Terry Kraft
   Navy Warfare Development Center
   Norfolk, VA

5. CAPT Steve Parode
   Navy Cyber Warfare Development Group
   Suitland, MD

6. Ms. Rosemary Wenchel
   Cyber Capabilities and Operations Support
   Washington, DC

7. Mr. Frank Sinkular
   Unified Cross Domain Management Office
   Adelphi, MD

8. CDR Arnie Brown
   Unified Cross Domain Management Office
   Adelphi, MD

9. Prof. James Bret Michael
   Naval Postgraduate School
   Arlington, VA

10. Prof. Doron Drusinsky
    Naval Postgraduate School
    Monterey, CA

11. Prof. George W. Dinolt
    Naval Postgraduate School
    Monterey, CA

12. Prof. Dan C. Boger
Naval Postgraduate School
Monterey, CA

13. Prof. Duminda Wijesekera
George Mason University
Fairfax, VA

14. Professor Man-Tak Shing
Naval Postgraduate School
Monterey, CA

15. Kathryn Hobbs, CAPT, USN (Ret.)
Commission on Veterans Issues
Olympia, WA

16. CDR Michael Schumann
Navy Cyber Warfare Development Group
Suitland, MD